Quality of Service of Crash-Recovery Failure Detectors

Tiejun Ma

Doctor of Philosophy

Laboratory for Foundations of Computer Science

School of Informatics

University of Edinburgh

2007

Abstract

This thesis presents the results of an investigation into the failure detection problem. We consider the specific case of the Quality of Service (QoS) of crash failure detection. In contrast to previous work, we address the crash failure detection problem when the monitored target is resilient and recovers after failure. To the best of our knowledge, this is the first work to provide an analysis of *crash-recovery* failure detection from the QoS perspective.

We develop a probabilistic model of the behavior of a *crash-recovery* target, i.e. one which has the ability to recover from the crash state. We show that the *fail-free* run and the *crash-stop* run are special cases of the *crash-recovery* run with mean time to failure (MTTF) approaching to infinity and mean time to recovery (MTTR) approaching to infinity, respectively. We extend the previously published QoS metrics to allow the measurement of the recovery speed, and the definition of the completeness property of a failure detector. Then, the impact of the dependability of the *crash-recovery* target on the QoS bounds for such a *crash-recovery* failure detector is analyzed using general dependability metrics, such as MTTF and MTTR, based on an approximate probabilistic model of the two-process failure detection system. Then according to our approximate model, we show how to estimate the failure detector's parameters to achieve a required QoS, based on Chen *et al*.'s NFD-S algorithm analytically, and how to execute the configuration procedure of this *crash-recovery* failure detector.

In order to make the failure detector adaptive to the target's *crash-recovery* behavior and enable the autonomy of the monitoring procedure, we propose two types of recovery detection protocols. One is a reliable recovery detection protocol, which can guarantee to detect each occurring failure and recovery by adopting persistent storage. The other is a lightweight recovery detection protocol, which does not guarantee to detect every failure and recovery but which reduces the system overhead. Both of these recovery detection protocols improve the completeness without reducing the other QoS aspects of a failure detector. In addition, we also demonstrate how to estimate the inputs, such as the dependability metrics, using the failure detector itself.

In order to evaluate our analytical work, we simulate the following failure detection al-

gorithms: the simple heartbeat timeout algorithm, the NFD-S algorithm and the NFD-S algorithm with the lightweight recovery detection protocol, for various values of MTTF and MTTR. The simulation results show that the dependability of a recoverable monitored target could have significant impact on the QoS of such a failure detector. This conforms well to our models and analysis. We show that in the case of reasonable long MTTF, the NFD-S algorithm with the lightweight recovery detection protocol exhibits better QoS than the NFD-S algorithm for the completeness of a *crash-recovery* failure detector, and similarly for other QoS metrics.

Keywords: Failure Detector, Failure Detection, QoS, Crash-Recovery, Fault Tolerance, Dependability, Web Services, Grid Computing.

Acknowledgements

I wish to express my deepest gratitude to my supervisors Dr. Stuart Anderson and Prof. Jane Hillston.

Stuart is my principle supervisor. I want to thank Stuart for accepting me into LFCS to study for my Ph.D and granting me the Ph.D scholarship. He has offered me an opportunity to get to know the dependability research and taught me scientific research skills. I greatly appreciate Stuart helping me find such a challenging research topic that I am really interested in and which has actually changed my life. His endless support and guidance has kept me going all these years and his great pedagogic talents enabled me to finally finish it.

Jane is my second supervisor. I am most grateful to Jane for providing me the highest quality of guidance with her profound knowledge throughout my Ph.D study. Jane's support has gone far beyond that of a top-notch supervisor, especially for the analysis work in my thesis, where she has given me valuable advice on mathematical issues. I have greatly appreciated her broad knowledge, clear thinking, and ideas. She has patiently and subtly trained me to organize my writing. I cannot have imagined having a better advisor and mentor for my Ph.D. Without her common-sense, knowledge and perceptiveness I could never have finished.

Thanks my examiners, Prof. Isi Mitrani of the Newcastle University and Dr. Mahesh Marina of the Edinburgh University, their suggestions of correction and revision help me improve the quality of this thesis.

I owe my warmest thanks to my colleagues Dr. Massimo Felici and Mr. Conrad Hughes for their valuable critique, advice and availability. Thanks Massimo providing me a chance to be a tutor of the Software Engineering and Object Programming course, which has given me plenty of practise in my teaching and communication skills.

I would also like to thank Prof. Alan Bandy, who always efficiently organized the e-Science group meetings during my first and second year Ph.D study, and set up a CoG discussion group which provided me a more regular chance to communicate with other researchers within the Informatics School.

I give special thanks to my friends Lin Yang and Bin Yang for actively organizing our private interest group meetings and sharing delicious food together while struggling through our Ph.D period. Thanks to Xibei Jia for kindly providing his solid knowledge of XML and Database. They were very important friends of mine during my years at the School of Informatics. Thanks to all of my friends for all the joy they bring to my life.

I wish to extend my thanks to the National e-Science Center for providing me with a great environment and facilities to carry out this work during my first year of Ph.D study.

Finally, I am deeply grateful to my father, mother, sister, brother-in-law and Xiaoli. Without their continued love, firm support and constant encouragement, the successful completion of this study could never have been possible. Especially thanks Xiaoli, for all your many valuable suggestions. Thank to everybody in my family for being there for me from the very beginning.

The following publications contain some parts of this thesis:

[1] T. Ma, J. Hillston, and S. Anderson. Evaluation of the QoS of Crash-Recovery Failure Detection. In *SAC '07: Proceedings of the ACM Symposium on Applied Computing (DADS Track)*. ACM, 2007.

[2] T. Ma, J. Hillston, and S. Anderson. On the Quality of Service of Crash-Recovery Failure Detectors. In *the Int. Conf. on Dependable Systems and Networks (DSN2007)*. IEEE Computer Society, June 2007.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Tiejun Ma*)

Dedicate to my father and mother.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Dependability is one of the most important issues for distributed systems. Since world-wide distributed systems have become more and more complicated, many fault-tolerance techniques have been adopted to improve system dependability. Dependable Infrastructure Grid Service (DIGS)[1] [4], one of the UK e-Science projects, aims to investigate dependability approaches for the service-oriented architecture and build strategies to enhance the service-oriented applications' reliability and availability. One of the key obstacles to achieving such a dependability goal is detecting occurred failures and handling faulty components promptly. The failure detection approach we have investigated is a subtopic of the DIGS project which brings together ideas of fault tolerance, reliability and dependability for the system infrastructures for large-scale distributed systems. The final aim of this failure detection topic is to provide theoretical and practical solutions to detect various types of failure with a satisfactory Quality of Service (QoS). In this thesis, we investigate the crash failure detection problem in detail based on the *crash-recovery* model. In general, the *crash-stop* failure is used to model a target that once it stops, it will never be repaired and become dead completely or when a problem occurred, it can be replaced by an another instance with

[1] This project is renamed as Dependable Service-Centric System (DCCS).

a new identifier. For the *crash-recovery* failure, it is usually used to model the target that after a failure occurs, it can be repaired, or has the ability of self-healing, then back to the correct state again. In theory, the *crash-recovery* failure model is mainly considered on the consensus and group membership problems [1, 34, 54, 69], in which the monitored process will crash by stop working and rejoin the member group after its recovery. In practice, in order to achieve high availability, self-repairing and self-healing mechanisms are widely adopted in fault-tolerant systems to achieve automatic recovery after the crash occurs. Particularly in middleware systems, there are many techniques and algorithms are proposed to achieve the self-repairing or self-healing goal, such as the connector-based self-healing system described in [32, 77] or the reflection technique adopted in [12] or the snapshot algorithms in [61, 65]. As we can see that the *crash-recovery* failure is quite common in many fault-tolerant systems. It needs to be considered as a frequently occurred failure type. However, compared with the *crash-stop* target, due to the state space size increasing of the *crash-recovery* target, QoS analysis of such a *crash-recovery* failure detector become more difficult in theory and practice. In a *fail-free* or *crash-stop* run, the state space size of the failure detector is two, which contains the *Trust-Alive* state and the *Suspect-Alive* state[2]. But in a *crash-recovery* run, the state space size is four, which contians the *Trust-Alive* state, the *Trust-Crash* state, the *Suspect-Alive* state and the *Suspect-Crash* state (details in Section 3.6.2). Therefore we focus on such a *crash-recovery* paradigm and study the failure detector's QoS based on a two-process failure detection pair system model.

1.2 Problem Statement

For global-scale distributed systems, various types of failure may occur during the execution. In this thesis, we address the QoS of the crash failure detection oracles as the first step. For crash failure detection, Fisher *et al.* in [38] show the impossibility of separating a crashed process and a very slow process in a pure asynchronous system, then Chandra and Toueg in [22] introduce the concept of unreliable crash failure detectors to

[2]For a *crash-stop* run, the *pre-crash* duration is mainly considered and the *pre-crash* duration of the *crash-stop* process is a long run.

detect the eventual crash behavior of a process. Following the previous work done by Chandra and Toueg in [22], Chen *et al.* in [24] give the definitions of the QoS metrics of failure detectors in terms of the *completeness* and the *accuracy* properties introduced in [22]. Furthermore, many researchers have drawn their attentions on the QoS of crash failure detectors' implementations and failure detection algorithms. However, most previous work on the QoS of crash failure detection is based on the *crash-stop* or the *fail-free* assumption and relies on predicting the liveness message transmission behavior and estimating a suitable timeout threshold to achieve a better QoS of crash failure detection. In contrast, we regard a crashed process or service as recoverable, since many fault-tolerance techniques can be adopted to achieve such recovery. For high-level applications, a more realistic crash failure model would be *crash-recovery*. However, the previous QoS analysis work and algorithms cannot adapt to the recovery of the monitored target. This is because the *fail-free* assumption assumes that failure does not occur. The *crash-stop* assumption assumes that there is only one failure and the monitoring procedure terminates once the occurred crash failure is detected. Furthermore, both of these two assumptions do not take the mistakes caused by the crash or recovery of the monitored target into consideration. In addition, in large-scale distributed systems, autonomic failure detection procedure is important. *Crash-stop* failure detectors cannot adapt to the *crash-recovery* target with a consistent view of the target's liveness. Thus, in this thesis, we mainly focus on such a *crash-recovery* paradigm. In a *crash-recovery* model, it is assumed that a system undergoes periodic crashes. During a crash period, the system is unable to service any requests or send any messages, which externally behaves like that the system is unreachable. The end of the crash period is marked by a recovery, after which the system returns normal service and its internal state is back to the same state before the crash failure occurs. In this thesis, we are concerned with the QoS offered by a failure detector monitoring such a system. We do this in the context of a two-process failure detection pair consisting of a failure detector and a monitored target process.

1.3 Thesis Contribution

In this thesis, we extend the previous study of *crash-stop* failure detection and consider the monitored target as *crash-recovery*. We study and model a *crash-recovery* target and its failure detector's probabilistic behavior. We extend the QoS metrics proposed in [24] to measure the recovery detection speed and the proportion of the detected failures of a *crash-recovery* failure detector. Then the impact of the dependability of the *crash-recovery* target on the QoS bounds for such a *crash-recovery* failure detector is analyzed by adopting general dependability metrics such as *mean time to failure* (MTTF) and *mean time to recovery* (MTTR). In addition, we analyze how to estimate the failure detector's parameters to achieve the QoS from a given set of requirements based on the NFD-S algorithm proposed by Chen *et al.* [24]. We also demonstrate how to estimate the inputs of a failure detector such as message delays, losses and the dependability metrics of a *crash-recovery* target, using the failure detector itself and demonstrate how to execute the configuration procedure of this *crash-recovery* failure detector.

In order to adapt to the recovery of the monitored target, we propose two types of recovery detection protocol, which can detect the occurrence of a recovery and estimate the recovery time. The first is a reliable recovery detection protocol, which guarantees to detect the occurrence of each failure and recovery by using persistent storage. The second is a lightweight protocol, which does not guarantee to detect every failure and recovery but which has a lower system overhead than our first protocol. Both of these recovery detection protocols can improve the proportion of failures detected without reducing other QoS aspects.

We simulate and evaluate the simple timeout algorithm, the NFD-S algorithm and the NFD-S algorithm with the lightweight recovery protocol. The simulation results match our analysis and show that the dependability of a *crash-recovery* monitored target will influence the QoS of such a failure detector. We compare the simulation results with the analytical results derived from our approximate model. The comparison shows that our analysis is valid and the simulation results satisfy the analytical QoS bounds: $T_D \leq T_D^U$, $E(T_{MR}) \geq T_{MR}^L$, $P_A \geq P_A^L$, $E(T_M) \leq T_M^U$, $E(T_{DR}) \leq T_{DR}^U$ and $E(R_{DF}) \geq R_{DF}^L$,

which match our modeling and analysis work in Chapters 3. Moreover, the proposed recovery detection protocols can enable the FDS to operate autonomously and improve the QoS of failure detectors in terms of the failure detection proportion for the highly consistent monitored target.

1.4 Thesis Outline

This thesis is structured as follows. In Chapter 2, first, we introduce some background of distributed systems briefly. Second, a literature survey of the previous work on the crash failure detection is presented. Third, we analyze and classify the possible types of failure that might occur and their detection strategies. Finally, based on our survey, we summarize various requirements for crash failure detectors and the previous work on crash failure detection.

In Chapter 3, we model a *crash-recovery* service and its failure detector's probabilistic behavior. We refine the *completeness* of a *crash-recovery* failure detector and extend the QoS metrics to measure the *completeness* and the recovery detection speed of such a failure detector. Then we show how to involve the general dependability metrics for an approximate analysis of the QoS of a failure detector and how to estimate the parameters and configure a *crash-recovery* failure detector to satisfy a given set of QoS requirements. Moreover, we discuss the impact of the dependability of the *crash-recovery* service on the QoS of failure detectors in detail.

In Chapter 4, the estimation of the input parameters of a *crash-recovery* failure detector is presented. We show how to estimate the message delay, message loss, MTTF and MTTR etc. in a *crash-recovery* run. Then, we propose two types of recovery detection protocol to improve the QoS and assist in the input parameter estimation of a failure detector.

In Chapter 5, we introduce the design of a failure detection simulator called FD-Simulator, which we adopt to evaluate our analysis work in Chapters 3 and 4. The proposed simulator can simulate either flat or hierarchical distributed failure detection frameworks and the *crash-recovery* behavior of the components within the simulated

frameworks. Based on this simulator, a simulation plan is proposed and the simulation cases are simulated. Then the simulation results are analyzed and we show that the dependability of a *crash-recovery* service has an impact on the QoS of a failure detector. The simulation results also show that the NFD-S algorithm with the lightweight recovery detection protocol performs better than the NFD-S algorithm in terms of the proportion of the detected crash failures, which implies that the recovery detection protocol can help a failure detector to adapt to the recovery behavior of the monitored target and improve the QoS of failure detectors. These results also indicate that recovery detection protocols can improve the adaptivity of *crash-recovery* failure detectors, which are particularly useful for highly consistent recoverable monitored targets.

Finally in Chapter 6, a brief summary of the thesis work is presented and some possible future directions are introduced.

Chapter 2

Failures and Mechanisms of Failure Detection

2.1 Introduction

The purpose of failure detection is to discover abnormal software behaviors. Recognizing the occurrence of failures is one of the most important steps towards achieving fault-tolerance and dependability. In this chapter, we introduce some related background knowledge and review existing crash failure detectors. The structure of this chapter is as follows: Section 2.2 and Section 2.3 contain the essential background related to this thesis. Section 2.2 introduces basic concepts of distributed systems, Web Services, Grid Computing, dependability and fault tolerance. Section 2.3 introduces the fundamental research work on unreliable failure detectors defined by Chandra and Toueg in [23] and QoS metrics defined by Chen *et al.* in [24]. Section 2.4-2.7 contain more details and discussions, which can be skipped for the first time reading. In Section 2.4, an overview of the previous work on crash failure detection algorithms and implementations is introduced. Section 2.5 introduces some failure detection service frameworks and applications. Section 2.6 introduces the classification of various types of failure. Finally, in Section 2.7, we discuss some desired features of failure detectors from designers' and users' viewpoints and summarize a range of requirements for the

design and implementation of failure detectors.

2.2 Background

2.2.1 Traditional Distributed Communication Paradigms

During the past decades, scientists and standardization organizations have made many efforts to enable geographically distributed applications to communicate with each other over networks. For example, socket communications and network protocols (e.g., TCP/IP, UDP) or CORBA [45] and DCOM [76] at a higher-level, that adopt remote procedure call (RPC) or remote method invocation (RMI). At the application-level, CORBA, DCOM and RMI can serialize, encapsulate and transmit the remote procedure call or invocation automatically to mask the lower level communication. All of these middleware communication paradigms greatly enhance the interactive ability of distributed applications developed in a mixture of languages based on different platforms and significantly reduce complexity and costs of application development. However, the platform and language dependency problems still remain to be completely solved.

2.2.2 Service-Oriented Architecture

Service-oriented architecture (SOA) has emerged over several years, and it has been well-accepted in the world of system design and development. In particular, for distributed systems, SOA has been widely adopted for its simplicity, comprehensibility, universality, effectiveness and integrity. Architectures such as DCOM and CORBA are all service-oriented. In general, SOA is a relatively abstract concept, which expresses an architectural style whose goal is to achieve loose coupling among interacting software components [47]. A service-oriented implementation is a collection of well-defined, self-contained software entities deployed by a service provider, which are accessible by service consumers via existing protocols.

2.2.3 Web Services

The Web Service architecture [14] is defined by the World Wide Web Consortium (W3C), which adopts SOA to publish functions over a network. In order to solve the interaction problem in traditional distributed systems, Web Services adopt XML-based definition languages and communication protocols at the application-level to ensure platform and language independence. The core standards of Web Services include Web Services Description Language (WSDL) [25] and Simple Object Access Protocol (SOAP) [15], which are defined by W3C. WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information, as a machine-processable format to describe the functions exposed by a Web Service [14]. The operation and messages in WSDL are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. SOAP is used to pass messages and exchange of information in a decentralized environment, typically conveyed using HTTP and XML to express the message payload. SOAP is an XML-based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined data types, and a convention for representing remote procedure calls and responses.

There are also many other standards such as Universal Description, Discovery and Integration (UDDI) [28] for service registration and discovery, Web Services Inspection Language (WS-Inspection) [7] which is another discovery specification and Web Service Level Agreement (WSLA) [64] which addresses service level management in a Web Services environment.

2.2.4 Grid Computing

Grid Computing has emerged as a brand-new research area in distributed computing. The concept of the Grid was first expressed by Ian Foster and Carl Kesselman in [40] in 1998 and enriched in [41]. Generally speaking, the Grid can be regarded as a global-

scale distributed system. The original purpose of Grid computing is to seamlessly share heterogeneous computational resources such as CPU power and storage capacity across dynamic virtual organizations, which can supply massive or even unlimited computing and storage capability. In recent years, Grid and Web Services standards have started to converge. The Grid community tried to merge Grid standards with Web Services in order to make them fully compatible and benefit each other. Now both adopt SOA as their foundation. The underlying heterogeneous infrastructures are masked by the service-oriented environment and applications can be deployed and made accessible in a distributed way as services.

2.2.5 Failures, Fault Tolerance and Dependability

Software and hardware may contain bugs and other internal or external errors that can make the run-time services unstable. A number of research papers [44, 43, 70] based on various types of computer system have shown that bugs are one of the most important reasons for system crashes. Thus faults are accepted as inevitable and may lead to a system failure. In this situation, fault-tolerance mechanisms are introduced for critical systems to improve the system survivability when failures occur. Roughly speaking, fault tolerance is the ability or the property to enable a system to continuously operate correctly when some abnormal internal or external events occur (e.g., failures). Many techniques have been proposed to achieve fault tolerance for various systems, such as adopting replications [10, 11] to introduce redundancies into a system, adopting checkpointing techniques [61, 65, 87] to snapshot the runtime information persistently, adopting rejuvenation [5, 60, 85] to achieve self-healing and using recovery [61, 72], reboot [17] or micro-reboot [18] to achieve re-birth. All of these techniques can enhance the system reliability, availability and consistency and usability etc. to some extent.

Dependability is one of the most important issues for computer systems, which is a complex attribute. Laprie *et al.* [63] define the concept of dependability as *the property of a computer system such that reliance can justifiably be placed on the service it delivers.* In addition, by recording the lifetime information of a system, the system's

dependability can be described quantitatively. Generally speaking, the dependability of a system can be measured according to the reliability, availability, consistency, usability and security etc. In order to simplify the measurements which are related to failure detection, here we only introduce reliability, availability and consistency, which are strongly related to the QoS of failure detectors (the relationships will be presented in the following chapters).

In [26], Knight and Strunk give a definition of software reliability and availability. We extend the definition of consistency as follows,

- Reliability: can be defined as the probability that the system will operate correctly in a specified operating environment up until time t ($t > 0$).
- Availability: can be defined as the probability that the system will be operational at time t.
- Consistency: can be defined as the probability that the system will return to normal operation correctly after a failure has occurred within a specified operating environment within time t.

These three metrics present different aspects of the system dependability. Generally, reliability presents how long a system will operate correctly and can be captured by *mean time to failure* (MTTF), which records the probability of a service to persist without a failure. Availability presents the probability that a system is accessible or reachable with correct operation at any time and can be captured by *mean time to failure* divided by *mean time between failure* ($\frac{\text{MTTF}}{\text{MTBF}}$). Consistency presents the ability of a system to recover from a failure state to the correct operation state and can be captured by *meantime to recovery* (MTTR), which records how fast a system recovers.

Furthermore, from the system design perspective, different systems might desire different aspects of dependability features, such as the highly reliable system which requires the system to be durable, the highly available system, which requires the system to be accessible with correct operation most of the time or the highly consistent system, which requires fast recovery of the system after failures occur. Thus the adoption of fault-tolerance mechanisms should adapt to dependability requirements.

2.2.6 Crash Failure

Crash failure is one of the most fundamental types of failure, which is traditionally regarded as a faulty process stopping performing its specification permanently. A crash failure happens when a correctly behaved service stops operating then remains inactive or can be repaired after some repair time. Generally speaking, there are two paradigms of the crash failure:

- *Crash-stop*: a correctly behaved service halts abnormally and stops forever after crashing.
- *Crash-recovery*: a service stops operating for a while then recovers to the correct state before it crashed.

In addition, the *fail-free* assumption is used quite often to simplify the study of the crash failure detection problem. This is because most monitored targets will be in the *alive* state for a long time before it crashes and such crash failures can be ignored as rare events in theoretical analysis in some occasions. Since *fail-free* is not an actual failure, we do not include *fail-free* as a failure type.

Basically, when we study crash failure, we assume that the crash failure is *fail-fast* [78], which means the service reaches the inactive state quickly. Sometimes, a *fail-silent* failure is regarded as similar to a crash failure, but more precisely, it is a muteness failure [35]. Actually, the crash failure is a particular case of the muteness failure, which will stop operating without generating any liveness messages when a failure occurs.

Various detection strategies can be adopted in detecting crash failures: *push-model* (receiving "I am alive message"), *pull-model* (querying "Are you alive") or *hybrid-model* (combining *pull* and *push*). If a service is capable of providing heartbeats or answering queries, crash failure is detectable by regularly checking the receipt of liveness messages. If the missing liveness messages are not received before the specified time threshold, the failure detector will suspect the liveness of the monitored target.

2.3 Crash Failure Detector Oracles

Detection of crash failure is crucial to many fundamental problems, such as solving consensus or group membership problems. Fisher *et al.* in [38] show the impossibility of solving distributed consensus problem in a pure asynchronous system, and Chandra and Toueg in [22] show that the weakest class of failure detectors to solve distributed consensus is $\diamondsuit\mathcal{S}$ (see Section 2.3.1), both of which mean that it is impossible to implement a reliable *crash-stop* failure detector in a pure asynchronous system. Their results show that even for the most fundamental *crash-stop* failure, detecting such a failure is still a knotty problem. In the following parts, we will introduce previous efforts and results in the context of crash failure detection.

2.3.1 Properties of Unreliable Failure Detectors

Chandra and Toueg in [22] first formally address the notion of failure detectors and introduce the concept of unreliable failure detectors in terms of the *completeness* and *accuracy* properties. The process failure type they considered is *crash-stop* failure presented in 2.2.6 and the properties of unreliable failure detectors are classified as follows:

- *Completeness*
  - *Strong completeness*: eventually every process that crashes is permanently suspected by every correct process.
  - *Weak completeness*: eventually every process that crashes is permanently suspected by some correct process.
- *Accuracy*
  - *Strong accuracy*: correct processes are never suspected.
  - *Weak accuracy*: some correct process is never suspected.
- *Eventual accuracy*

  - *Eventual strong accuracy*: there is a time after which no correct processes is suspected by any correct process.
  - *Eventual weak accuracy*: there is a time after which some correct process is never suspected by any correct process.

With respect to the satisfaction of the above definitions, a failure detector can be categorized according to the following classes.

- $\mathcal{P}$: the set of *Perfect Failure Detectors* that satisfy the *strong completeness* and the *strong accuracy* properties
- $\mathcal{S}$: the set of *Strong Failure Detectors* that satisfy the *strong completeness* and the *weak accuracy* properties
- $\mathcal{W}$: the set of *Weak Failure Detectors* that satisfy the *weak completeness* and the *weak accuracy* properties
- $\mathcal{Q}$: the set of *Failure Detectors* that satisfy the *weak completeness* and the *strong accuracy* properties
- $\Diamond\mathcal{P}$: the set of *Eventually Perfect Failure Detectors* that satisfy the *strong completeness* and the *eventual strong accuracy* properties
- $\Diamond\mathcal{S}$: the set of *Eventually Strong Failure Detectors* that satisfy the *strong completeness* and the *eventual weak accuracy* properties
- $\Diamond\mathcal{W}$: the set of *Eventually Weak Failure Detectors* that satisfy the *weak completeness* and the *eventual weak accuracy* properties
- $\Diamond\mathcal{Q}$: the set of *Eventually Failure Detectors* that satisfy the *weak completeness* and the *eventual strong accuracy* properties

From the above definitions, the properties of a failure detector can be captured by its *completeness* and *accuracy*, which provides a precise way to measure the conditions that are needed to solve consensus problems by using failure detectors and establish the classification of crash failure detectors. Then the weakest failure detector to solve a certain distributed agreement problem under a given system environment can be deduced accurately according to Chandra and Toueg's classification. However, using

	Accuracy			
Completeness	Strong	Weak	Eventual Strong	Eventual Weak
Strong	*Perfect* $\mathcal{P}$	*Strong* $\mathcal{S}$	*Eventually Perfect* $\Diamond\mathcal{P}$	*Eventually Strong* $\Diamond\mathcal{S}$
Weak	$\mathcal{Q}$	*Weak* $\mathcal{W}$	$\Diamond\mathcal{Q}$	*Eventually Weak* $\Diamond\mathcal{W}$

Table 2.1: Classes of Failure Detectors Defined in Terms of Accuracy and Completeness

only *completeness* and *accuracy* properties are not enough to measure how well a failure detector can satisfy the performance requirements. Consequently, in order to measure the QoS of crash failure detectors, Chen *et al.* in [24] give a complex set of QoS metrics to measure the performance of crash failure detectors. These are discussed in the following section.

2.3.2 QoS Metrics of Crash Failure Detectors

In [24], Chen *et al.* propose a set of QoS metrics to measure the *completeness*, *accuracy* and *speed* of unreliable failure detectors. In order to formally define the QoS metrics, Chen *et al.* define state transitions of a failure detector: when a failure detector monitors a monitored process, at any time, the failure detector's state either trusts or suspects the monitored process's liveness. If a failure detector transfers from a *Trust* state to a *Suspect* state, then a *S-transition* occurs; if a failure detector transfers from a *Suspect* state to a *Trust* state then a *T-transition* occurs. Fig. 2.1 shows QoS metrics and state transitions of the failure detector as in [24]. Here is a short introduction to the main QoS metrics and formulas to compute the failure detector parameters.

QoS Metrics:

- **Detection time** (T_D): the elapsed time from when the monitored process crashes until the monitoring process suspects the monitored process permanently (the final *S-transition* occurs).

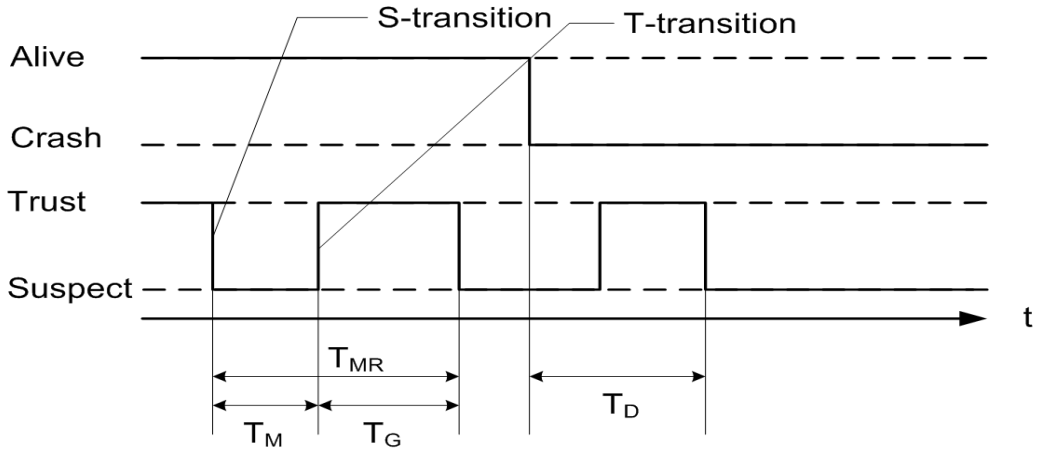


Figure 2.1: The QoS Metrics

- **Mistake recurrence time** (T_{MR}): this is the time between the i and $i+1$ mistake occurrences (*S-transition* i to *S-transition* $i+1$), where $i \geq 1$.
- **Mistake duration** (T_M): the time to correct a mistake from the suspect state (*S-transition* to *T-transition*).
- **Average mistake rate** (λ_M): the number of mistakes per unit time that a failure detector makes ($\lambda_M = \frac{1}{E(T_{MR})}$).
- **Good period duration** (T_G): the duration for which the failure detector maintains the correct state information. (*T-transition* to next S-*transition*, $T_G = T_{MR} - T_M$).
- **Query accuracy probability** (P_A): the probability that the failure detector can give correct state information for the monitored process at an arbitrary time ($P_A = \frac{E(T_G)}{E(T_{MR})}$).
- **Forward good period duration** (T_{FG}): the duration for which at a random time, the monitor process trusts the monitored process until the next *S-transition* occurs ($T_{FG} = \frac{[1+\frac{V(T_G)}{E(T_G)^2}]E(T_G)}{2}$).

In the work of Chen *et al.*, they are only interested in the system before a crash of the monitored process. Moreover, it is assumed that this eventuality is as distant that the system reaches an equilibrium behavior before the crash occurs. This has been clarified

by the addition of an expanded explanation of the assumptions inherent in Chen *et al*'s framework.

In addition, Chen *et al.* show that, the QoS of failure detectors can be captured by T_D, T_M, T_{MR}, which can be the primary QoS metrics. All of the other QoS metrics can be derived from these three QoS metrics. Furthermore, Chen *et al.* also design three *push-style* algorithms for systems with synchronized and unsynchronized clocks, which will be introduced in Section 2.4.

2.4 Crash Failure Detection Algorithms

crash failure is one of the most important types of failure that need to be discovered. Most crash failure detectors are based on detecting the liveness message sent by the monitored target. In this section, we will survey the previous research work on the crash failure detectors' design and implementation.

Chen *et al.*'s Algorithms

In [24], Chen *et al.* consider the failure detection model which contains two processes—the failure detector (q) and the monitored process (p). Chen *et al.* propose three *push-style* algorithms as *crash-stop* failure detectors, one for systems with synchronized clocks (NFD-S) and the other two for systems with unsynchronized clocks (NFD-U and NFD-E). The authors show how to estimate the parameters of the failure detector (heartbeat interval (η) and shift of freshness point δ) for NFD-S, NFD-U and NFD-E, respectively (see pseudocode Algorithm 3, Algorithm 4 and Algorithm 5 in Appendix A), when the QoS requirements—the upper bound detection time, the upper bound of mistake duration and the lower bound of mistake recurrence time $(T_D^U,\ T_M^U,\ T_{MR}^L)$—are given. The authors consider the message transmission as probabilistic, which can be captured by the probability of message loss (p_L) and the probability of message delay within x time duration ($Pr(D \leq x)$). The algorithm details are introduced below: the monitored process p periodically sends heartbeat messages $m_1,\ m_2,\ m_3,\ \cdots$ to q every η time units. Every heartbeat message m_i is tagged with its sequence number i. Then, σ_i denotes the sending time of message m_i. The monitoring

process q shifts the σ_is forward by δ to obtain the sequence of times $\tau_1 < \tau_2 < \tau_3 \cdots$, where $\tau_i = \sigma_i + \delta$. Process q uses the τ_is (the *freshness points*) and the times it receives heartbeat messages to determine whether to trust or suspect p during the time period $[\tau_i, \tau_{i+1})$. At time τ_i, q checks whether it has received some message m_j with $j \geq i$. If so, q trusts p during the entire period $[\tau_i, \tau_{i+1})$. If not, q starts suspecting p. If, at some time before τ_{i+1}, q receives some message m_j with $j \geq 1$, then q starts trusting p from that time until τ_{i+1}. If, by time τ_{i+1}, q has not received any message m_j with $j \geq i$, then q suspects p during the entire period $[\tau_i, \tau_{i+1})$. This procedure is repeated for every time period. From time τ_i to τ_{i+1}, messages m_j with $j \geq i$ are still fresh. Therefore the NFD-S algorithm is characterized by the following property: q trusts p at time t if and only if q receives a message that is still fresh at time t.

In order to formally analyze the QoS bounds of the NFD-S algorithm, the following definitions and propositions are defined and proved as below:

Definition:

1. k: for any $i \geq 1$, let k be the smallest integer such that, for all $j \geq i+k$, m_j is sent at or after time τ_i.
2. For any $i \geq 1$, let $p_j(x)$ be the probability that the failure detector does not receive message m_{i+j} by time $\tau_i + x$, for every $j \geq 0$ and every $x \geq 0$; let p_0= $p_0(0)$.
3. For any $i \geq 2$, let q_0 be the probability that the failure detector receives message m_{i-1} before time τ_i.
4. For any $i \geq 1$, let $u(x)$ be the probability that the failure detector suspects the monitored process at time $\tau_i + x$, for every $x \in [0,\ \eta)$.
5. p_s: for any $i \geq 2$, let p_s be the probability that an *S-transition* occurs at time τ_i.

Proposition:

1. $k = \lceil timeout/\eta \rceil$.
2. For all $j \geq 0$ and for all $x \geq 0$,

$$p_j(x) = (p_L + (1-p_L)Pr(D > \delta + x - j\eta)).$$

3. $q_0 = (1-p_L) \cdot Pr(D < \delta + \eta)$.
4. For all $x \in [0,\ \eta)$,

$$u(x) = \prod_{j=0}^{k} p_j(x).$$

5. $p_s = q_0 \cdot u(0)$.

For NFD-S, the QoS metrics can be estimated as follows:

- The detection time: $T_D \leq \delta + \eta$.
- The average mistake recurrence: $E(T_{MR}) = \dfrac{\eta}{p_s}$.
- The average mistake duration: $E(T_M) = \dfrac{\int_0^\eta u(x)dx}{p_s}$.
- The query accuracy probability: $P_A = 1 - E(T_M)/E(T_{MR}) = 1 - \dfrac{\int_0^\eta u(x)dx}{\eta}$.

According to the given QoS requirements ($T_D^U,\ T_M^U,\ T_{MR}^L$), and the QoS of message transmission ($p_L,\ Pr(D \leq x)$), the parameters of the failure detector can be estimated by using the following inequalities: $\delta + \eta \leq T_D^U$, $\frac{\eta}{p_s} \geq T_{MR}^L$ and $\frac{\int_0^\eta u(x)dx}{p_s} \leq T_M^U$.

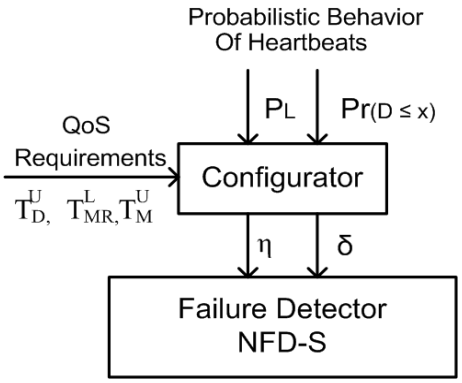


Figure 2.2: The NFD-S Algorithm Configuration

Fig. 2.2 shows the configuration procedure of the NFD-S algorithm. The calculation procedure of η and δ can be performed by the following three steps:

Step 1: Compute $q_0' = (1 - p_L)Pr(D < T_D^U)$ and let $\eta_{max} = q_0' T_M^U$. If $\eta_{max} = 0$, then output "QoS cannot achieved" and stop; else continue.

Step 2: Let

$$f(\eta) = \frac{\eta}{q_0' \prod_{j=1}^{\lceil T_D^U/\eta \rceil - 1} [p_L + (1 - p_L)Pr(D > T_D^U - j\eta)]}.$$

Find the largest $\eta \leq \eta_{max}$, such that $f(\eta) \geq T_{MR}^L$, which can be done by adopting binary search.

Step 3: Set $\delta = T_D^U - \eta$ then output η and δ.

In addition, the authors also show how to deal with unknown message transmission behavior by using the average message delay $E(D)$ and the variance of message delay $V(D)$. With $E(D)$ and V(D), $Pr(D > t)$ can be estimated by using the *One-Sided Inequality* of probability theory as follows,

$$Pr(D > t) \leq \frac{V(D)}{V(D) + (t - E(D))^2}, \text{ for all } t > E(D).$$

Then from the above inequality, the computation procedure of η and δ of the NFD-S algorithm can be revised as follows,

Step 1: Compute $\gamma' = \frac{(1-p_L)(T_D^U - E(D))^2}{V(D)+(T_D^U - E(D))^2}$ and let $\eta_{max} = min(\gamma' T_M^U, T_D^U - E(D))$. If $\eta_{max} = 0$, then output "QoS cannot satisfied" and stop; else continue;

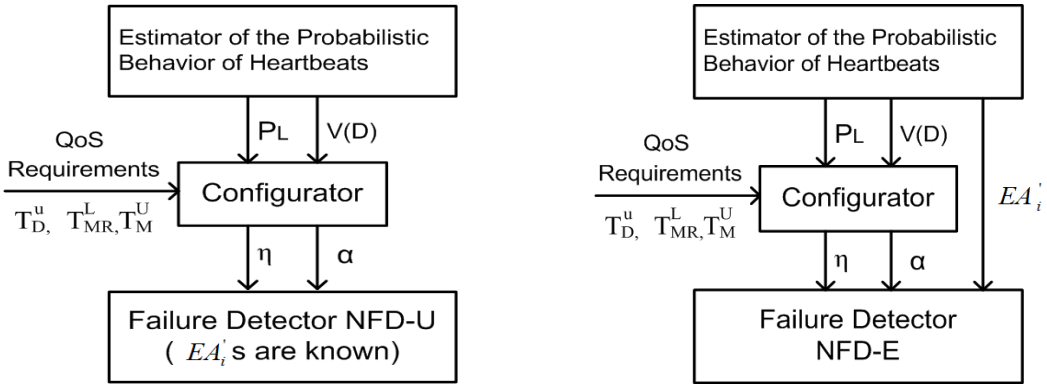
Step 2: Let

$$f(\eta) = \eta \cdot \prod_{j=1}^{\lceil \frac{T_D^U - E(D)}{\eta} \rceil - 1} \frac{V(D) + (T_D^U - E(D) - j\eta)^2}{V(D) + p_L(T_D^U - E(D) - j\eta)^2}$$

and find the largest $\eta \leq \eta_{max}$ such that $f(\eta) \geq T_{MR}^L$;

Step 3: Set $\delta = T_D^U - \eta$ and output η and δ.

For NFD-U, the algorithm assumes that the expected arrival time of the ith heartbeat message (EA_i) is known. It differs from the NFD-S algorithm only in the way they set the *freshness points* τ_is. In NFD-S, $\tau_i = \sigma_i + \delta$, but in NFD-U, $\tau_i = EA_i + \alpha = \sigma_i + E(D) + \alpha$ ($\delta = \alpha + E(D)$). Thus the absolute upper bound of detection speed T_D^U is substituted by $T_D^u + E(D)$, where T_D^u is the given required upper bound of detection speed and $E(D)$ is average message delay. Therefore, the parameters configuration procedure of the NFD-U algorithm is shown in Fig. 2.3(a). and it can be done in the



(a) NFD-U (b) NFD-E

Figure 2.3: NFD-U and NFD-E Algorithms Configuration

following three steps:

Step 1: Compute $\gamma' = \frac{(1-p_L)(T_D^u)^2}{V(D)+(T_D^u)^2}$ and let $\eta_{max} = min(\gamma' T_M^U, T_D^u)$. If $\eta_{max} = 0$, then output "QoS cannot satisfied" and stop; else continue;

Step 2: Let

$$f(\eta) = \eta \cdot \prod_{j=1}^{\lceil \frac{T_D^u}{\eta} \rceil - 1} \frac{V(D) + (T_D^u - j\eta)^2}{V(D) + p_L(T_D^u - j\eta)^2}$$

and find the largest $\eta \leq \eta_{max}$ such that $f(\eta) \geq T_{MR}^L$;

Step 3: Set $\alpha = T_D^u - \eta$ and output η and α.

The parameters computation and configuration procedure of the NFD-E algorithm are very similar to the NFD-U algorithm (see Fig 2.3(b)). But the NFD-E algorithm regards the ith expected arrival time of heartbeat message (EA_i) as an unknown parameter and the expected heartbeat arrival time can be estimated by the following equation:

$$EA_{\ell+1} \approx \frac{1}{n}(\sum_{i=1}^{n} A_i' - \eta s_i) + (\ell+1)\eta,$$

where s_1, ..., s_n are the sequence numbers of the heartbeat messages and A_1', ..., A_n' are the receiving times according to the monitor process's local clock.

Jacobson's Estimation

In [56], Jacobson studies the network communication delay problem and proposes an estimation method to estimate such delay. In Jacobson's model, the behavior of the system is not constant. A safety margin is used to adapt the estimation bias, which uses the error in the last estimation. Therefore, the expected delay of the next message can be estimated by using the previous estimation and the weighted previous estimation error.

Bertier *et al.*'s Algorithm

In [9], Bertier *et al.* propose a *push-style Eventually Perfect* ($\Diamond\mathcal{P}$) failure detection algorithm (see pseudocode Algorithm 8 in Appendix A) for partially synchronous systems. Bertier *et al.*'s failure detector requires the *strong completeness* and *eventual strong accuracy* properties (see Section 2.3.1). In [9], every pair of the failure detector and the monitored process is assumed to be connected by a channel, which provides reliable communication. This failure detector's parameter estimation method adopts Chen *et al.* estimation of the message expected arrival time [24] and Jacboson's estimation of safety margin [56].

Bertier's failure detector has two layers: the first layer is the basic failure detection service, which can estimate the parameters of the failure detector that compromise between the number of false detections and the failure detection speed. The second layer is designed for adapting the application specific needs according to the failure detection service provided by the first layer.

Fetzer *et al.*'s Algorithm

In [37], Fetzer *et al.* propose a *pull-style* failure detection protocol (see pseudocode Algorithm 6 in Appendix A) to detect *crash-stop* failure. The protocol allows processes to monitor each other and assumes that every pair of processes is connected by a reliable communication channel. The failure detection messages (queries) are performed in a lazy style to save the network load, which means only when these processes are not communicating with each other, the failure detection protocol is used to verify the liveness of monitored processes. The protocol provides the following primitives to every process:

- sending a message to another process.
- receiving a message from another process.
- querying another process.

Each process p_i manages two arrays: one array contains the sending times of the messages sent by p_i to another process p_j whose acknowledgement has not yet been received by p_i (initially empty); the other array contains the maximum round-trip time of the messages that p_i has sent to p_j and that have been acknowledged (initially 0). When the system is running, the process p_i sends application messages attached with some control information to another process p_j. When p_j receives an application message, it will send an acknowledgement message back to p_i. When an acknowledgement message is received by p_i, the round-trip delay will be computed and the maximum round-trip time will be recorded. If an application message has already been sent to p_j but the acknowledgement message has not been received by p_i yet, the failure detector's output is still *no_suspect*. But p_i will send a query message to p_j and whether the failure detector will output a suspect will depend on the maximum round-trip delay which has been recorded. This lazy style failure detection protocol minimizes the cost

of message communication. However the detection speed aspect was not taken into consideration in the original paper.

Overall, when this failure detection protocol is used in a partial synchronous system, the *eventual completeness* and the *eventual accuracy* properties defined in [22] can be satisfied, which means it can be regarded as a $\Diamond\mathcal{P}$ failure detector.

Nunes and Jansch-Pôrto's Algorithms

In [68], Nunes and Jansch-Pôrto propose a *pull-style crash-stop* failure detector and evaluate the QoS of this *pull-style* failure detector with different combinations of three round-trip communication delay (*rtt*) predictors and two timeout safety margins. The adopted predictors are:

- MEAN: the next *rtt* is based on the population average.
- WINMEAN: the next *rtt* follows the average of the last *n* samples.
- ARIMA: *the Auto-Regressive Intergrade Moving Average*, which is based on non-stationary time series modeling.

The two adopted safety margins are the prediction error-based margin (*peb*) that adapts its value each time when a message is received and the network load has varied (the same as the Jacobson's estimation in [56]); the confidence interval-based margin (*cib*) assumes that the predictor appropriately models the round-trip communication delay and the prediction error is considered as a white noise.

The evaluation results of the above predictors and safety margins show that the combination of predictor-margin is important to achieve a good QoS, but there is no universal best solution to achieve best detection speed and best accuracy of the failure detector. Different combinations could achieve best QoS in some aspects but not all aspects. The choice of combination will depend on the user's QoS requirements. If detection speed is more important, a margin-based on the prediction error (*peb*) should be combined with an accurate predictor, but a conservative margin (*cib*) or a constant margin is better combined with the average-based predictor; if accuracy is more important, the use of an inaccurate predictor (e.g., MEAN) should combine with a prediction error-based margin, an accurate predictor should combine with a small conservative or constant

margin. The evaluation results also show that the combination of MEAN-*peb* exhibits more accuracy but ARIMA-*cib* exhibits a better balance between accuracy and speed.

Falai and Bondavalli's Experimental Evaluation

Similar to Nunes and Jansch-Pôrto's work in [68], Falai and Bondavalli in [36] describe and interpret some experiments based on a *push-style crash-stop* failure detector , which combine several heartbeat messages delay predictors and timeout safety margins. Furthermore they evaluate the QoS for the different alternatives. The experimental system contains one failure detector and one monitored process. The communication between the failure detection pair is through a wide-area network. The adopted predictors are LAST, MEAN, WINMEAN(N), ARIMA, LPF (the predicted value is the exponential smoothing of the observations, which is a special case of ARIMA). Falai and Bondavalli adopt two safety margins—confidence interval-based margin (SM_{CI}) and the previous error-based margin (SM_{JAC}) — proposed by Nunes and Jansch-Pôrto in [68] with three different values (high, middle, low) for each safety margin respectively.

The evaluation results show that the MEAN predictor exhibits the slowest detection speed with all of the safety margins. LPF combined with SM_{CI} obtains best detection speed. LAST combined with SM_{JAC} obtains the best accuracy. Their evaluation results also draw the following conclusions:

- First, adopting an accurate predictor does not imply better accuracy of the failure detector. The combination of a predictor and a safety margin is the key to achieve accuracy.
- Second, fast detection implies lower mistake reoccurrence time. If mistake reoccurrence time is more important, then the safety margin should be increased, which implies that the detection speed will slow down as well.
- Third, an accurate predictor with a constant safety margin, or a less accurate predictor assisted by an adaptable safety margin, can achieve better accuracy of the failure detector.
- Fourth, LAST+SM_{JAC} achieves the lowest complexity and system overhead ac-

cording to the calculation costs. But it offers a quite good detection speed and accuracy property of the failure detector.

- Finally, a trade-off certainly exists between the failure detector's accuracy and speed. There is no universal solution to achieve the best speed and accuracy at the same time.

Sotoma and Madeira's Algorithm

In [81], Sotoma and Madeira extend Chen *et al.*'s failure detector in the presence of heartbeat message loss bursts. In [81], the independence property of each heartbeat message transmission (used in [24]) is assumed not to hold any more. The authors adopt and extend Sanneck's message loss-length model [53], which defines a finite Markov chain model for message loss length with $m+1$ states (see Fig. 2.4). The model contains a random variable X, which presents the number of consecutive message losses. If $X = 0$, it means there is no message lost; if $X = k$ $(0 < k < m)$ it means exactly k messages have been lost.

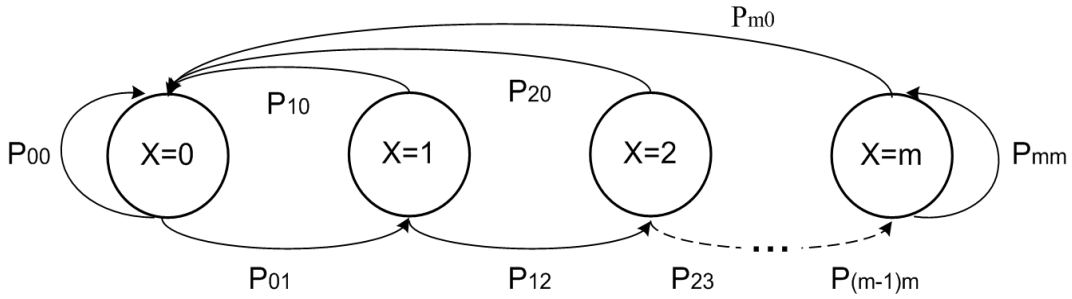


Figure 2.4: The Sotoma and Madeira's Message Loss Model Base on the Sanneck's Model with Limited State Space

When the parameters of the failure detector are computed, the dependencies between the consecutive liveness messages are taken into consideration and the relevant probability calculation follows Table 1 in [81].

The simulation and evaluation of Sotoma and Madeira's failure detector are based on Chen *et al.*'s NFD-S algorithm with the same system conditions as in [24]. Their simulation results show that by adopting this Markov model and revised calculation steps, the failure detector can adapt better to the occurrence of heartbeat message loss

bursts and achieve a better QoS of failure detectors.

Hayashibara *et al.*'s Algorithms

In [51], Hayashibara *et al.* propose an accrual failure detector and describe an adaptive *push-style* failure detection implementation called φ failure detector (see pseudocode Algorithm 7 in Appendix A). Instead of outputting a binary state decision value (Trust or Suspect) for the monitored process, accrual failure detectors output suspicion level ($susp\_level_p(t) \geq 0$) information on a continuous scale and the higher the value, the higher the chance that the monitored process has crashed. The accrual failure detector can sample the arrival time of heartbeats and maintain a sliding window of the most recent samples. The sliding window is used to estimate the arrival time of the next heartbeat. The probabilistic distribution of future heartbeat messages is estimated by using the distribution of history samples. Then a value φ, with a scale that changes dynamically to match recent network conditions, is computed by learning from the distribution information. Fig. 2.5 shows the architecture of accrual failure detectors. Instead of making decisions in the failure detector, accrual failure detectors leave the liveness interpretation of the monitored process to higher level applications, which brings more flexibility to the failure detector and associated applications (see Fig.2.5).

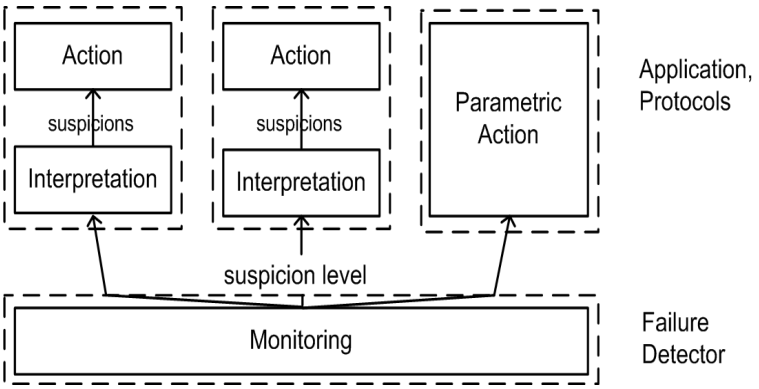


Figure 2.5: The Structure of Accrual Failure Detectors

In addition to *strong completeness* and *eventual strong accuracy*, the authors also constrain the $susp\_level_p(t)$ to the following properties:

- *Asymptotic completeness*: if the monitored process is faulty, the suspicion level $susp\_level_p(t)$ tends to infinity as time goes to infinity.

- *Eventual monotony*: if the monitored process is faulty, there is a time after which $susp\_level_p(t)$ is monotonically increasing.
- *Upper bound*: the monitored process is correct if and only if $susp\_level_p(t)$ has an upper bound over an infinite execution.
- *Reset*: if the monitored process is correct, then for any time t_0, $susp\_level_p(t) = 0$, for some time $t > 0$.

Instead of using *S-transition* and *T-transition* as in [24], two dynamic thresholds T_{high} and T_{low} are proposed, which are both initialized greater than 0. Then the transitions are defined as:

- *S-transition*: whenever the value of $susp\_level_p(t)$ crosses the upper threshold T_{high} upward, q updates the value of T_{high} to $T_{high} + 1$, and begins to suspect p (or continues to suspect p if it does so already).
- *T-transition*: whenever the value of $susp\_level_p(t)$ crosses the lower threshold T_{low} downward, q updates the value of T_{low} to that of T_{high}, and stops suspecting p.

The proposed algorithm for an accurate failure detector is similar to that of Fetzer *et al.*'s failure detection protocol in [37] and the φ failure detector is proposed as an implementation of the abstraction of an accrual failure detector, where φ represents the suspicion level of the accrual failure detector. φ can be scaled dynamically according to the message transmission condition.

In addition to the φ failure detector, a κ failure detector [49] is proposed to solve the consecutive message loss problem. The κ failure detector uses a function named *contribution function* $c(t)$ to determine the evolution of the confidence of each heartbeat message. Each missed heartbeat contributes to raising the level of suspicion of the failure detector.

Finally, the experimental results show that the φ failure detector performs equally well as other adaptive failure detectors with an improved flexibility and the κ failure detector can be more adaptive to consecutive liveness message loss behavior.

2.5 Failure Detection Service Implementations

In addition to the theoretical algorithms presented in the previous section, there are some failure detection applications designed from an engineering and implementation perspective, which focus more on the adaptation, scalability and efficiency aspects.

Renesse *et al.*'s Failure Detector

In [71], Renesse *et al.* propose a *push-style crash-stop* failure detection service based on the random gossip protocol (many-to-many), which provides probabilistic accuracy of the detected failure. The main purpose of this gossip protocol is to guarantee the high probability of accuracy as well as to balance the network bandwidth consumption. But as a trade-off, the detection speed is sacrificed. The detection time increases as $O(n \log n)$, where n is the number of member processes. Moreover, this gossip protocol is resilient against both transient message loss, permanent network partitions, and host failures. The authors introduce two protocols as follows:

- Basic protocol: a member forwards network information to randomly chosen members. Each member maintains a list for each known member's address and an integer which is going to be used for failure detection; each member occasionally broadcasts its list in order to be located initially and also to recover from network partitions; each member also maintains, for each other member in the list, the last time that its corresponding heartbeat counter has increased. If the heartbeat counter has not increased for more than a threshold time, then the member is considered failed. After a member is considered failed, it will not be removed until another threshold time, which is larger than the threshold of the faulty judgment time, to ensure the accuracy.

- Multilevel Gossiping: the multilevel gossiping protocol is proposed to cope with the efficiency and scalability problems of the basic protocol in a large-scale system. In this protocol, the lengths of subnetworks and host numbers for each domain are attached to each gossip message so that the connectivity between the subnetworks is discovered by the higher-level gossip messages. This means that few messages will cross subnetworks and even fewer will cross domains. With

concern to the topology of the network, the multilevel protocol can reduce the number of messages that flood through routers, which connect different subnetworks.

Stelling *et al.*'s Failure Detector

In [82], Stelling *et al.* propose a *hybrid-style crash-stop* hierarchical failure detection framework called the Globus Heartbeat Monitor (HBM), which can be deployed within asynchronous distributed systems as multilevel failure detection services (see Fig. 2.6) to detect a Grid Service's crash. The framework adopts two-layer monitoring. For the lower-layer, the authors use local monitors to detect service failures within subdomains to avoid cross-network communication messages. The framework provides *local monitors* which adopt a simple *pull-style* algorithm (inquiry) to monitor the services within the local domain. It provides higher-level *data collectors* which can gather and maintain a global view of the failure information from *local monitors*' heartbeat messages.

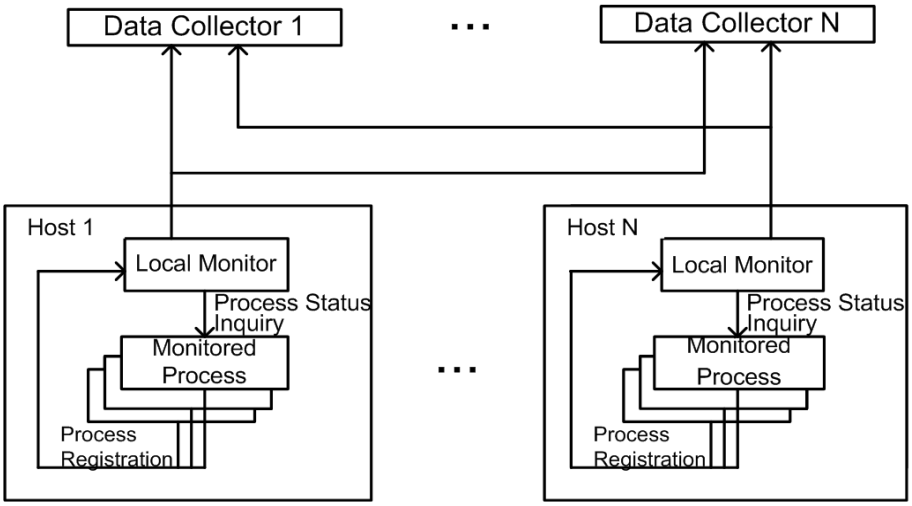


Figure 2.6: The Architecture of The Globus Fault Detection Service

The framework provides three basic components. The first one is called the *local monitor*, which is responsible for monitoring the computer on which it runs, as well as the selected processes on that computer. The second one is the client registration API, which can be adopted by each monitored application to inject the ability of registering

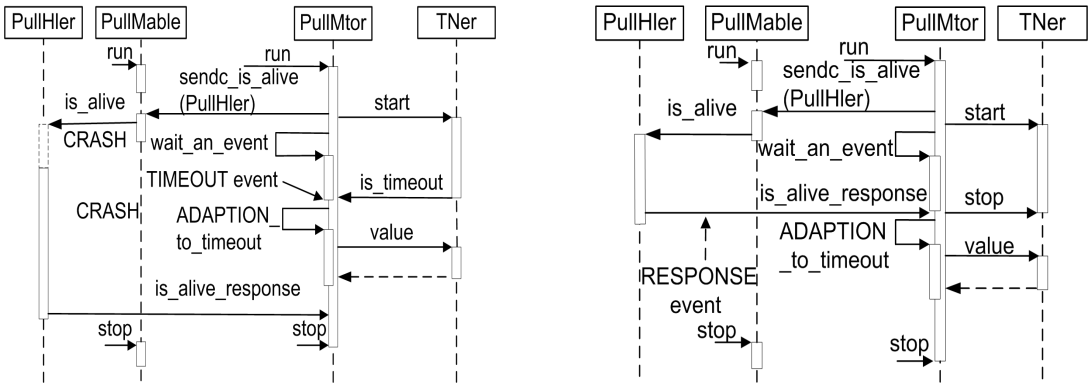
itself with the *local monitor* and sending heartbeats. The client registration API also provides the *local monitor* with the ability to send heartbeat and local monitoring information messages to *data collectors*. The last component is the callback-based data collection API, which allows applications to register a function to be called when a failure event occurs. The *data collectors* can keep track of all registered processes and record whenever a heartbeat arrives. When an expected heartbeat is missing, the *data collectors* can generate callbacks for the missing heartbeat for the individual process. In addition, the callbacks can also be issued when some interesting events occur.

Overall, the whole framework is trying to achieve QoS of crash failure detection as well as to balance the system overhead. The framework also separates the semantics of failure detection from higher level reactions according to the occurred failures, all of which provides more scalability and flexibility for the failure detection framework.

Sotoma and Madeira's CORBA Failure Detectors

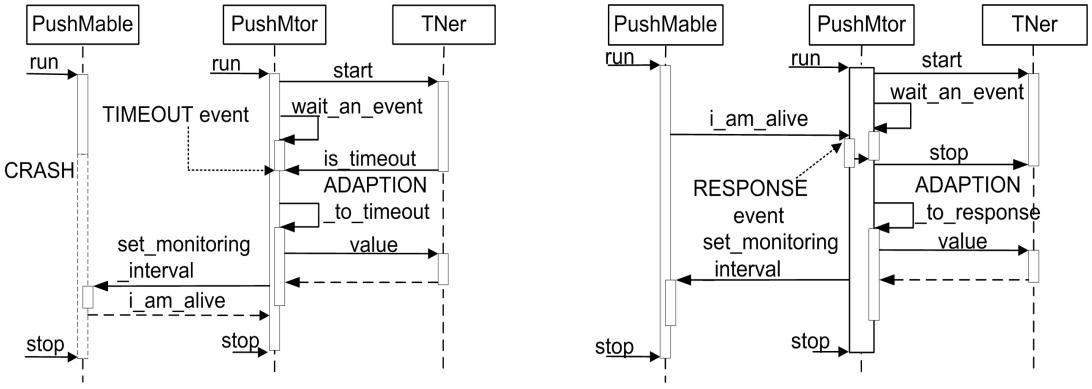
In [80], Sotoma and Madeira propose a CORBA implementation of adaptive *crash-stop* failure detectors. Their algorithms are designed for asynchronous distributed systems with reliable communication in both a *pull* model and a *push* model to achieve a better failure detection speed and minimize the discrepancy of erroneous suspicion. The adaptation algorithms are based on adjusting the monitoring interval and timeout according to the history—the average of message inter-arrival time and the average ratio of current inter-arrival over the average of the historical inter-arrival times, which are used to estimate the future message's inter-arrival time and the future system workload oscillation, respectively. Fig. 2.7 shows the sequence diagrams of the *pull-style* adaptive failure detector. Fig. 2.8 shows the sequence diagrams of the *push-style* adaptive failure detector.

The experimental results show that the *push-style* and the *pull-style* adaptive CORBA implementations exhibit some good features. First, in both of the proposed *push-style* and *pull-style* adaptation algorithms, it is not necessary to specify an initial timeout or a monitoring interval. Second, in terms of the proposed adaptation algorithms, the *push-style* implementation is better than the *pull-style* implementation when the system workload variation is fast. Third, the *pull-style* algorithm is almost free of influences by



(a) When a timeout event occurs (b) When a timeout event does not occurs

Figure 2.7: The Sequence Diagrams of the Pull Adaptation Algorithm



(a) When a timeout event occurs (b) When a timeout event does not occurs

Figure 2.8: The Sequence Diagram of the Push Adaptation Algorithm

the monitoring interval, but *push-style* algorithm is largely affected by the monitoring interval. Finally, both algorithms can adapt to changing system conditions and *push-style* algorithm works better because of the smaller monitoring interval value.

Das *et al.*'s Failure Detector

In [31], Das *et al.* propose a *pull-style* failure detector named scalable weakly-consistent infection-style process group membership protocol (SWIM) for *crash-stop* failure detection. The protocol is asynchronous-based and the communication is unreliable. The SWIM failure detector has two components: a failure detector component, which

detects failures of members and a dissemination component, which propagates information about the state changes of members. SWIM works in a peer-to-peer way. Each group member (M_i) periodically chooses some random members (e.g., M_j) in its membership list and sends ping messages to probe the liveness of these other members. If the monitored member receives a ping message, then it will send an acknowledgement message back. If the acknowledgement message has not been received within a given timeout duration, then M_i will send a ping request message to some non-faulty group members to ping the M_j as well. (see Fig. 2.9) If none of the direct or indirect acknowledgement messages of M_j are received by M_i then a failure of M_j is declared.

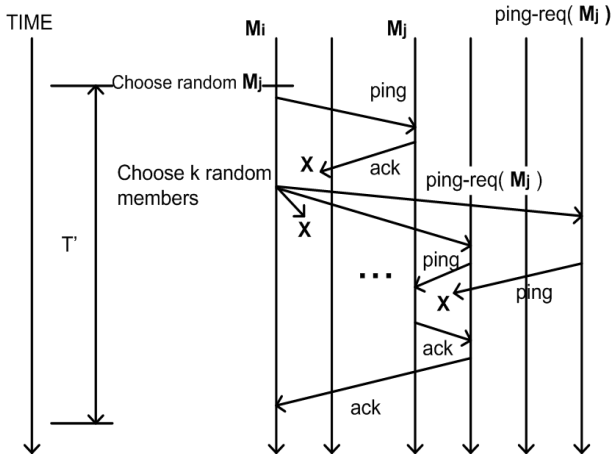


Figure 2.9: The SWIM Failure Detector

The SWIM failure detector exhibits some good characteristics. First, SWIM can provide deterministic failure detection speed, which is unlike some other gossip style failure detectors such as [71]. Second, the expected message load of a SWIM member does not vary with group size. Third, membership joins, drop outs and failures are propagated via piggybacks on ping or acknowledgement messages, which leads to fast dissemination.

2.6 Classification of Failure types

In order to detect occurred failures, having knowledge of failure types and their behaviors is an important issue. Laprie *et al.* in [63] define the fundamental concept of a failure as: *a failure occurs when an actual running system deviates from its specified behavior*. Several proposals for failure classification are presented from different perspectives, such as [13, 63]. But none of them introduce the failures as a hierarchical classification with chained behavior. Therefore in addition to the crash failure mentioned in Section 2.2.6, we also introduce the following possible failures that might occur within the software system and their relationships to help the readers have a further understanding about failures.

2.6.1 Muteness Failure

Muteness failures are malicious failures in which a process stops sending algorithm messages, but might continue to send other messages [35]. When a muteness failure occurs, the service will stop executing its designed features but might still be able to generate liveness messages. Thus such failures cannot be detected by crash failure detectors. Muteness failure is a particular case of omission failure, which fails to send some messages but not all messages.

Detecting muteness failure could be application-specific. Generally, message timeout-based detection strategies can be adopted. For example, adopting the muteness failure detection algorithm in [35], which proposes a protocol that forces the monitored service to send "I-am-not-mute" messages to the muteness failure detector periodically when the service is not mute, but stop sending such messages when a muteness failure occurs. Then if the muteness failure detector does not receive such a message within the given timeout threshold, the failure detector will assume that a muteness failure has occurred.

2.6.2 Timing Failure

Timing failure occurs when a service's response lies outside the specified time interval [63]. For example, if the service-hosting machine or network is overloaded, or some other resources on which the service depends are overloaded, then the service response might be delayed and a timing failure might occur.

The detection of timing failure should be based on the specified deadline or time constraints. In order to detect a timing failure, recording the time when the conversation between a service pair starts can be adopted. If the service instance cannot return the answer before the specified deadline, it is regarded as a timing failure. Moreover, there are more sophisticated timing failure detectors such as the one reported in [3] which uses group communication to detect timing failure in a quasi-synchronous (incompletely synchronous) system; or the Timely Computing Base (TCB) model in [20], which can deal with timeliness requirements without synchronized clocks.

2.6.3 Omission Failure

When a service fails to send a response or receive a message, an omission failure occurs. Omission failure externally behaves as a communication failure, which will cause message transmission to fail.

The simplest way to detect omission failures is to enable the service to provide failure information. If the service can throw a *fail-to-send-response* or *fail-to-receive-message -exception* or send this information to the failure detector then the failure is regarded as an omission failure. Otherwise, it might be considered as a timing failure or a false crash failure. Dolev *et al.* [34] adopt Chandra and Toueg's definitions of failure detectors [22] (see Section 2.3.1) and design a protocol to detect omission failure by adopting consensus. Their results show that the failure detector they designed can solve $\lceil \frac{n-1}{2} \rceil$ consensus problems within asynchronous systems.

2.6.4 QoS Failure

The QoS delivered by a service is an important concern. A service, even if it provides a correct result, might still fail to meet the consumers' desired level of service. A service exhibits a QoS failure when some observable behavior of the service fails to satisfy a specified property by the service consumer. This property may be specified by a certain level of QoS constraints. For example, 95% confidence that the mean time to get results is smaller than 10 seconds, assuming that initially we are 99% confident of this property.

QoS failure can be tracked by matching the given QoS specification with the QoS delivered by the service. If the QoS of the service cannot satisfy the given specification, then it is regarded as a QoS failure.

2.6.5 Response Failure

Response failure occurs when a service response is incorrect. In general, response failures can be separated into two types. The first type is value failure: the response value is wrong; the second type is state transition failure: the service deviates from the correct flow of control [63].

To detect value failure, voting algorithms can be adopted if multiple service replication is deployed. To detect state transition failure, the service design specification should be available to check whether a service has deviated from its expected state or not.

2.6.6 Partial Failure

For a composed application, a component failure may result in a partial failure of the composed service. Identifying such a partial service failure still remains challenging. Here we regard a component of a service as atomic and consider dependencies among these components. Failure of a component might potentially cause other failures of the composed service or the failure of the composing procedure.

For a composed service, due to service internal fault-tolerance policies, partial failure might not be visible externally by a failure detector, which only observes the composed service. In order to discover such partial failures, sensors must be implemented at the atomic component level to track the status information of each atomic component of a composed service. The implementation of the sensor for a component should be based on the failure mode that the sensor is concerned with. (For example, a sensor designed for tracking crash failure can adopt heartbeat detection.)

2.6.7 Composition Failure

Service composition is an important characteristic of Web Services. Any service partial failure or unmatched composition requirements would result in a service composition failure.

To detect such failures in a composing service, each composition step should be checked and tested. If the current composition procedure is verified without any mistake having occurred and the composition conditions are satisfied, then proceed to the next step; otherwise a composition failure might have occurred.

2.6.8 Byzantine Failure

The Byzantine failure is also sometimes called the arbitrary failure. It means a service may behave in an arbitrary manner, produce arbitrary responses at arbitrary time [63]. It is the most complicated failure to detect. According to the detection, Byzantine failures can be separated into *undetectable* and *detectable* failures [59]. *Undetectable* Byzantine failures refer to failures that are unobservable by other processes based on the messages they receive or failures that are undiagnosable. *Detectable* Byzantine failures have two categories:

1. Commission (Response) failures: the service does not behave correctly according to its semantics.

2. Omission failure: the service behaves correctly but fails to send or receive messages.

One possible solution of Byzantine failure detection is adopting consensus algorithms. For example, several redundant service replications may run simultaneously. After all of them have finished their work, then they start to form a consensus. If a majority of them return the same result, then send the answer to the customer. To achieve K fault tolerance, $3K+1$ service replications are needed [62]. In the worst case, the K faulty services may send incorrect values, or incorrectly represent the values of others, but the remaining $2K+1$ services can still return the same correct answer. Therefore, the customer can still get the right result. Obviously, Byzantine failure detection is costly and consensus might potentially cause timing failures. Whether to adopt consensus or not should depend on the desired service QoS from the consumers' viewpoint. For example, in certain environments, such as aero control systems or medical diagnosis systems, any mistake is unacceptable. In such situations, consensus voting should be adopted. But for some other applications, such as an online video service, or a file transfer service, some mistakes can be tolerated within the application, and time is more important. It is not necessary to adopt expensive voting algorithms in these cases.

2.6.9 The Relationship Analysis of Various Types of Failure

In previous sections, we descried various possible types of failure. Actually, they are not isolated. Most of them are associated with each other. Clarifying their relationships will improve our understanding of service failure behaviors as well as optimize failure detector design and implementation. According to the classification of Byzantine failure given in [35], we extend the model and summarize the relationship of various types of failure (see Fig. 2.10), which might be exhibited by the composed service externally as certain failures. The design of a failure detector for a particular type of failure will completely depend on the failure type concerned. Moreover, the distributed communication environment will also have impact on the algorithm to adopt. In Fig. 2.10, the difficulty, complexity and costs to detect the possible types of failure from bottom

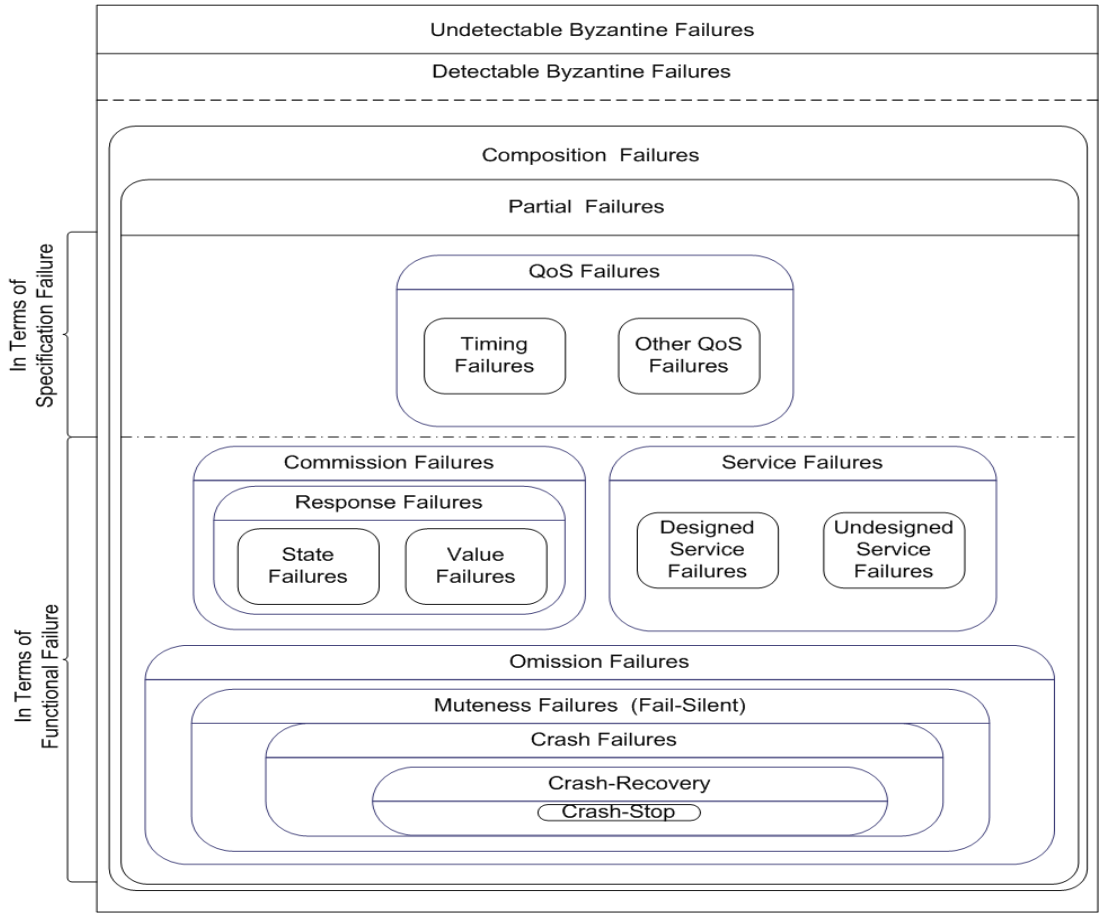


Figure 2.10: A Classification of Failure Types

to top will dramatically increase and become even worse when the communication of the distributed network is unreliable. Therefore to deliver a satisfactory QoS of an affordable failure detection service will be a great challenge. Within the partial failure frame, the failure types in the bottom layer, will affect the service availability or liveness. The relationship between the failure types is inclusion (e.g., *crash-stop* failure is a particular type of *crash-recovery* failure with the recovery time approaching infinity or the recovery rate approaching zero); the failure types in the middle layer will affect the validity of the service, which means a live service might not be valid in terms of the service semantics; the failure types in the top layer will affect the acceptability from service consumers' perspective, which means a valid service might not be a

satisfactory service in terms of QoS requirements.

According to the presented failures' classification, we separate the failure detection into three aspects, the first and basic aspect is to detect the liveness or availability of a service; the second aspect is to check the validity of a service; the third aspect is to verify the satisfactory behavior of a service. In addition, the failures within the first and the second layers in Fig. 2.10 can be regarded as functional failures, which will cause an application to behave improperly. From Fig. 2.10 we can see that the complex failure semantic and their chained relationship make failure detection never a trivial work. Exactly detecting the type and the location of an occurred failure quickly and accurately is difficult. In this thesis, we will mainly focus on crash failure detection problems at the bottom of in Fig. 2.10. In the following sections, we will introduce crash failure detector oracles and the related research work in detail.

2.7 Summary of Failure Detectors

In this chapter, various types of failure and their detection mechanisms have been classified and studied. The most fundamental failure—crash failure—has been surveyed in detail. Crash failure detection problems have been extensively investigated over the past decades. Some theoretical failure detectors, such as [1, 2, 22, 21] focus on the classification and design of crash failure detectors to solve certain distributed agreement problems. Some failure detection algorithms are designed with more accurate liveness message delay prediction, more suitable timeout duration or more proper combination of them to achieve a better QoS, such as [9, 24, 36, 37, 68, 81]. Others, such as [31, 33, 46, 48, 51, 71, 82], also make efforts to improve the scalability, adaptivity or efficiency, or to balance the system overhead aspects or to deal with network partition problems.

Overall, based on the surveyed literature, we summarize the requirements of designing and implementing crash failure detectors from various perspectives as follows:

- The Quality of Service: for any failure detector or failure detection framework, the expected QoS requirements should be satisfied, such as *completeness*, *accu-*

racy or failure detection speed requirements.

- Adaptivity: a failure detector or failure detection framework should quickly adapt (reconfigure) to the changing conditions of its runtime environment, such as network condition, or QoS requirements.
- Scalability: a failure detector or failure detection framework should scale well when the complexity of the monitoring environment changes. (e.g., monitoring group size varies, system size scales up or down)
- Efficiency: according to the achieved QoS, the complexity of running a failure detector or failure detection framework should be relatively low.
- Flexibility: a failure detector or failure detection framework should be easily adopted by various types of application.
- Low system overhead: a failure detection framework should not consume too many CPU cycles, disk IO or network bandwidth.
- Autonomous: once a failure detector or failure detection framework are deployed, they should run automatically to adapt to the failure and the recovery behaviors of monitored member targets. When a service is created, the monitoring should start automatically. When a service is terminated, the failure detector should discover this termination rather than regard it as a crash failure. For a flat or hierarchical failure detection framework, failure detectors should also be discoverable by other failure detectors as monitored targets.
- Dependability: a failure detector or failure detection framework should be highly reliable, available and consistent. Failure might happen within a failure detector. Single point of failure should be avoided within a failure detection framework. A failure detector or failure detection framework should be highly consistent to cope with the crash and recovery behavior of any components within the framework.
- Heterogeneity: it should be possible to deploy a failure detector or failure detection framework on heterogeneous platforms. This is because failure detectors might be deployed in different locations within one global distributed system

and might have to work on various types of platform.

- Information Propagation: the monitoring or failure detection information should be easily accessible and highly consistent.

Overall, from the survey studies and the requirements analysis in this chapter, we can see that failure detectors can be designed from various perspectives. There are no universal solutions to solve all failure detection problems. Thus designing or implementing a failure detector or a failure detection framework must associate with some requirements for an application-specific implementation within a particular environment. For crash failure detectors' design, many efforts have been made to address different failure detection issues. The unreliable failure detectors proposed by Chandra and Toueg [22] show the class of failure detectors to solve a certain set of agreement problems. But these failure detectors are abstract modules and cannot indicate how well the problem can be solved quantitatively. Chen *et al.* [24] address this problem and extend the concept of the unreliable failure detectors with a set of QoS metrics. In addition, the failure detectors proposed by Chen *et al.* [24] provide the QoS guarantee feature in terms of the given QoS requirements. However, the failure detectors proposed by Chen *et al.* are based on some strong assumptions, that are not very adaptive and scalable. For example, the monitored target is assumed to be *crash-stop* or *fail-free*, each message transmission is independent without any dependency; the failure detector parameters are estimated without considering the dynamic change of message delays and losses. The failure detectors proposed by Nunes and Jansch-Pôrto [68] address the dynamic message behavior problem and give some more complex message delay and loss estimation methods. Such failure detectors can be more adaptive in terms of the network condition. Falai and Bondavalli in [36] extend Nunes and Jansch-Pôrto's estimation methods and evaluate them in more detail. Their results clearly show that the trade-off exists between QoS metrics. The failure detector designer should be aware which QoS aspect is most important and then select the message delay and timeout estimation methods appropriately. Sotoma and Madeira [81] extend Chen *et al.*'s NFD-S algorithm without the independent message transmission assumption, but for other aspects the problems remains the same.

Nevertheless, all of the above failure detectors only can give a binary decision of the

monitored target's liveness. This results in a inflexible drawback for the failure detectors used by multiple applications. The failure detectors proposed by [51] address this flexibility problem. The accurate failure detectors Hayashibara *et al* designed can give a probabilistic confidence of the monitored target's liveness and let the application decide whether to trust the liveness of the monitored target or suspect. But the authors regard the crashes as rare events and the calculation of the failure detector's suspicion level is heavily based on a normal distribution. Such an assumption is also strong. In addition, the above introduced failure detectors only contain one failure detection pair. They are not scalable and flexible for large-scale distributed systems. Failure detectors such as [31, 71, 82] are proposed to solve the scalability and flexibility problems by adopting gossip protocols, group membership algorithms, or designing hierarchical frameworks. But in order to achieve the scalability or flexibility requirement, some other aspects are sacrificed. For example, most gossip failure detectors cannot deliver the fast detection speed property; the failure detectors adopting group membership protocols will generate more messages, which are more complex and less efficient compared to the paired failure detectors.

Furthermore, most existing crash failure detectors are based on the *crash-stop* or even *fail-free* assumption and the autonomous aspect is still untreated. These *crash-stop* or even *fail-free* failure detectors cannot adapt to the highly consistent *crash-recovery* monitored targets. But as we discussed in Section 1.1, in reality, such *crash-recovery* of monitored targets naturally exists within highly dynamic distributed systems. Thus we target this issue and try to fill in this gap in theory and practice in Chapters 3 and 4.

Chapter 3

Stochastic Modeling of A Crash-Recovery Service and Its FDS

3.1 Introduction

The Quality of Service (QoS) of crash failure detection is a widely studied topic [24, 36, 50, 51, 68, 81]. Most of the previous work, such as the papers we mentioned above, on this topic is based on the *crash-stop* or *fail-free* assumption. In this chapter, we investigate and model a *crash-recovery* target service (CR-TS), which has the ability to recover from the crash state. We analyze the QoS bounds for such a *crash-recovery* Failure Detection Service (FDS). Our results show that the dependability metrics of the CR-TS will have an impact on the QoS of the FDS. First, we introduce some background stochastic theory in Section 3.2, which will be used in this chapter. Then a brief survey of the related work is presented in Section 3.3. In Section 3.4, we model a repairable service as an alternating renewal process and the general dependability metrics such as reliability and availability are adopted for further analysis. In Section 3.5, we model the message transmission as channel-based message communication. In Section 3.6, we approximately model a *crash-recovery* FDS and analyze the possible mistakes in a *crash-recovery* run. We extend the QoS model of a *crash-stop* failure detector proposed by Chen *et al.* [24] to adapt to a *crash-recovery*

FDS. In addition to considering the QoS of liveness message transmission, we involve more factors in terms of the CR-TS dependability characteristics, which could affect the QoS of the FDS as well. We adopt the NFD-S algorithm proposed in [24], and analyze how to estimate the QoS bounds of a FDS based on the NFD-S algorithm in a *crash-recovery* run. In Section 3.7, we show how to configure the FDS to satisfy the required QoS in a *crash-recovery* run. Furthermore, in Section 3.8, we discuss a possible optimization of the QoS bounds estimation. In addition, the impact of the dependability of a *crash-recovery* service on the QoS of failure detectors is analyzed in Section 3.9. Finally, in Section 3.10, a brief summary of this chapter is presented.

3.2 Stochastic Theory Background

The purpose of this section is to present some general background and briefly summarize some results from the theory of stochastic processes which we will need in the following sections. Such theory is much more extensive than what we present here and can be found in many textbooks. For example we refer the reader to [57, 58, 67]. Most definitions and results in this part are due to [29, 58, 67]. Readers who are familiar with stochastic theory can skip this part and continue with the following sections without any difficulty.

Stochastic processes play an important role in various fields (economics, biology, reliability, etc.). In the mathematics of probability, a stochastic process is a random function, whose parameter t is often time.

Definition 3.2.1. *Let* $(\Omega,\ \mathcal{F},\ \boldsymbol{P})$ *denote a probability space, where* Ω *is the sample space,* $\mathcal{F}$ *is a* σ*-algebra of subsets of* Ω *and* $\boldsymbol{P}$ *is a probability measure on* $\mathcal{F}$*. A stochastic process* $X := \{X_t : t \in \boldsymbol{T}\}$ *is a parameterized collection of random variables with index set* $\boldsymbol{T}$*. For each fixed* $\omega \in \Omega$*, the function*

$$t \rightarrow X_t(\omega);\ t \in \boldsymbol{T}$$

is called a trajectory. When $\boldsymbol{T}$ *is discrete, then X is called stochastic process in discrete time; when* $\boldsymbol{T}$ *is an interval or half line of the real line, or the whole real line, then X is called stochastic process in continuous time.*

In our work, we will analyse a stochastic process termed a *renewal process*, which is used to model independent identically distributed occurrences. A renewal process is a generalization of the Poisson process. Generally speaking, the Poisson process is a continuous-time process on the positive integers which has independent identically distributed holding times at each integer i (exponentially distributed) before advancing (with probability 1) to the next integer $i+1$. We may define a renewal process to be the same thing, except that the holding times take on a more general distribution. Mathematically, a renewal process is a sequence of independent identically distributed positive random variables, which are not all zero, with probability 1.

Definition 3.2.2. *Let $X_1,\ X_2,\ \cdots$ be independent identically distributed positive and real-valued random variables and define the partial sum*

$$Z_n = X_1 + X_2 + \cdots + X_n.$$

Then the stochastic processes $\{Z_1,\ Z_2,\ \cdots Z_n\}$ is said to be a renewal denoted by

$$M(t) = \max\{n : Z_n \leq t\},\ t > 0.$$

Renewal processes have been widely used in solving problems in reliability theory. The following developments are made in the framework of reliability. When a failure occurs, let us assume that:

- the failed item is replaced or repaired, in such a way that it will be as good as new;
- the successive lifetimes are independent random variables distributed according to the probability density function $f(t)$.

The simple definition of a renewal process indicates that many types of stochastic process can be described as renewal. It is often the case that a complex stochastic model has one or more embedded renewal processes. This fact is basic to the idea of regeneration. The time instant at which the system is renewed is called a *regeneration point*. If we consider a non-exponential failure distribution $F(t)$ as a renewal process, then not every time instant is a regeneration point; only those time instants where renewals take place are regeneration points. For example, if the renewals take place at

time $t_i,\ i = 1,\ 2,\ \cdots$, then the points $\{t_1,\ t_2,\ \cdots\}$ are regeneration points whereas any point $x \in (t_{i-1},\ t_i)$ for all i is a non-regeneration point. Alternating renewal processes are special types of renewal processes, which the renewal interval has two alternating "up" and "down" states. It will be seen that formulating problems in terms of alternating renewal processes provides a powerful conceptual framework for dealing with important aspects of the behavior of complicated stochastic processes. A formal definition of an alternating renewal process is presented below.

Definition 3.2.3. *Let $\{x_i,\ i \geq 1\}$ and $\{y_i,\ i \geq 1\}$ denote sequences of independent and identically distributed non-negative random variables, but with x_i and y_i not necessarily independent. Let x and y denote generic random variables for x_i and y_i respectively; and let $P\{x > 0\} > 0$ and $P\{y > 0\} > 0$, so that $E|x| > 0$ and $E|y| > 0$. Define $z_i = x_i + y_i$ and let z denote a generic z_i. Further, define $s_n = \sum_{i=1}^{n} z_i$ and $s_0 = 0$ with probability 1, and let $\{n(t),\ t > 0\} = \sup\{n|s_n \leq t\}$. Then the counting process $\{n(t),\ t \geq 0\}$ is called an alternating renewal process, z is called the renewal interval, and s_n is called the time of the nth renewal.*

In the later part of this chapter, we adopt stochastic modeling and renewal theory to model a monitored service within a distributed system as an alternating renewal process (details in Section 3.4). The general dependability metrics, *mean time to failure* (MTTF) and *mean time to repair* (MTTR) are used as the basis for an alternating renewal process. MTTF is regarded as the mean of random variables, which presents the "up" time of the service; MTTR is regarded as the mean of random variables, which presents the "down" time of the service. Together they form one period of a "up−down" alternating renewal process and the mean of a period is called *mean time between failures* (MTBF).

3.3 Related Work

In [24], Chen *et al.* propose a set of QoS metrics to measure the *accuracy* and *speed* of a failure detector. Their model contains a pair of processes: one is the monitor process,

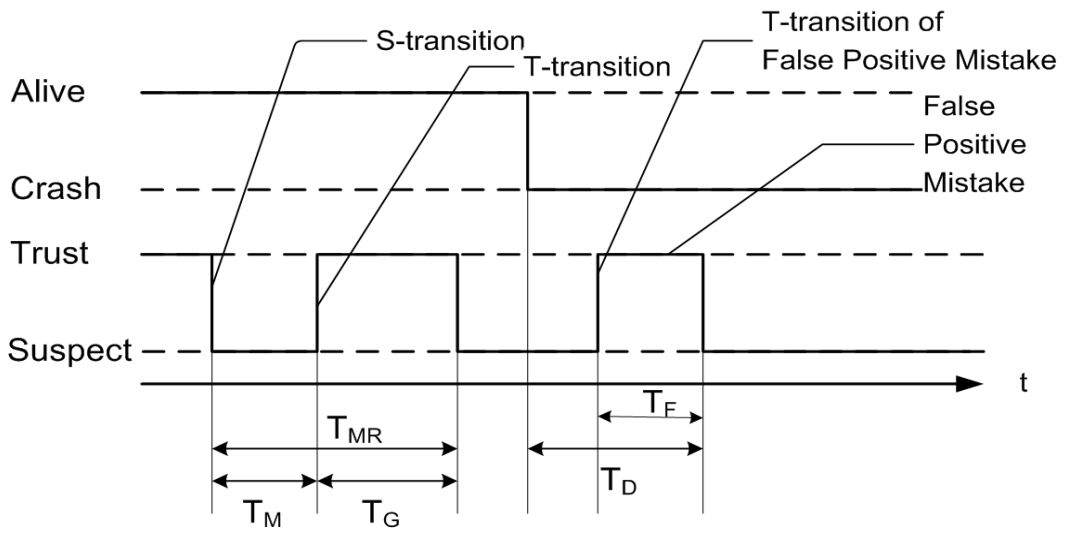


Figure 3.1: The QoS Metrics without Considering False Positive Mistakes

the other is the monitored process and there is only one crash during the monitoring duration. The analysis is based on two separate stages of failure detection: the *pre-crash* stage, which is a *fail-free* run; the *post-crash* stage, which is a *crash-stop* run when the monitoring procedure will be terminated. In order formally to define the QoS metrics, [24] defines state transitions of a failure detector monitoring a target process under the *fail-free* assumption. At any time, the failure detector's state is either *Trust* or *Suspect* with respect to the monitored process's liveness. If a failure detector moves from a *Trust* state to a *Suspect* state then an *S-transition* occurs; if the failure detector moves from a *Suspect* state to a *Trust* state then a *T-transition* occurs. Fig. 3.1 shows the state transitions of the failure detector and the QoS metrics. In terms of the transitions defined above and the *fail-free* assumption, Chen *et al.* define the following QoS metrics for a failure detector: failure detection time (T_D), mistake recurrence time (T_{MR}), mistake duration (T_M), average mistake rate (λ_M), good period duration (T_G), query accuracy probability (P_A) and forward good period duration (T_{FG}) (a more detailed introduction is given in Section 2.3.2).

Additionally in [24], they present three *push-style* algorithms, one for clock synchronized systems (NFD-S) and the other two for clock unsynchronized systems (NFD-U and NFD-E). They also show how to estimate the failure detector parameters (heart-

beat interval η and shift of freshness point δ[1]) according to a given QoS specification for each of the above algorithms (details in Section 2.4).

One limitation of the framework of [24] is that they disregard *false positive* mistakes. For example, they assume that after the $i+1$ heartbeat is sent, the monitored process crashes. Fig. 3.1 shows the *false positive* mistake in the state transition diagram, which has duration T_F.

Some recent research has extended the QoS work of [24] in a number of ways. For example, [9, 36, 68, 81] refine the model with different probabilistic message delay and loss estimation methods. Meanwhile, others, such as [46, 50, 51, 71, 82] focus on the scalability and adaptivity of crash failure detection. But all of these papers are based on eventual *crash-stop* behavior of the monitored process or the *fail-free* assumption. *Crash-recovery* failure detectors have been considered by several groups, e.g., [1, 34, 54, 69]. However, each of these papers proposes failure detectors to solve consensus problems rather than focusing on the QoS of the failure detector itself. In [1], the monitored process is characterized as *always-up*, *eventually-up*, *eventually-down* or *unstable*. A process which crashes and recovers infinitely many times is regarded as unstable. But *crash-recovery* looping behavior exists for most systems. From the perspective of stochastic theory, *crash-recovery* behavior can be regarded as a regenerative process, in which the probabilistic live and recovery time are not zero. In the following sections, we will analyze such a *crash-recovery* paradigm and its failure detector from a QoS perspective.

3.4 Crash-Recovery Service

3.4.1 The Crash-Recovery Service Modeling

For a *crash-recovery* target service (CR-TS), we consider the service might crash at arbitrary time and take some time to be repaired and restart again after it fails. Let us take $\mathcal{S}$ be the state space of a stochastic process $Z := \{Z(t), t \geq 0\}$, where Z is a CR-

[1] In order to simplify the expression, δ is replaced by *timeout* in this chapter.

TS's lifetime then $\mathcal{S}$ can be regarded as a set which has two states: *Alive* and *Crash*. Thus the state space of the CR-TS is defined as $\mathcal{S} := \{Alive, Crash\}$ and the CR-TS can periodically switch between these two states. A transition occurs when the state of the CR-TS changes. Fig.3.2 shows the state transitions of a CR-TS, where a *C-transition* occurs when the state of the CR-TS switches from the *Alive* state to the *Crash* state; an *R-transition* occurs when the state of the CR-TS switch from the *Crash* state to the *Alive* state. (see Fig. 3.2)

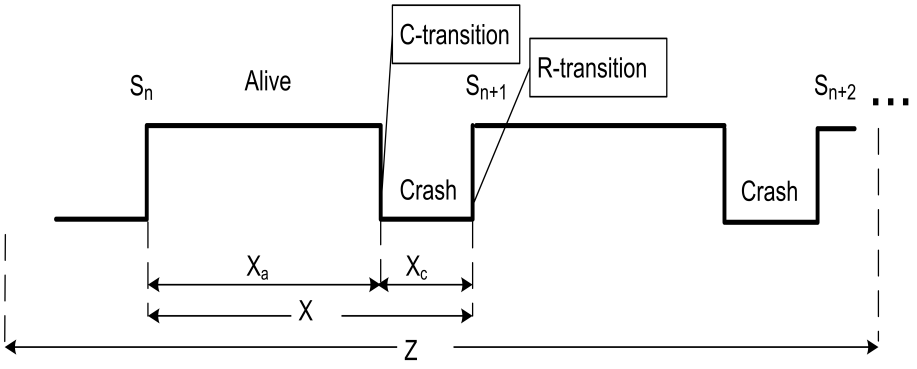


Figure 3.2: Crash-Recovery Service Modeling

Assumption 3.4.1. *If the service's recovery is treated as a restart, then the CR-TS's lifetime Z is a regenerative process.*

Assumption 3.4.1 will be used in the following parts. This is because obviously for the CR-TS, which will periodically crash and recover, there must exist a time-point sequence, $S_1,\ S_2,\ \cdots S_n,\ \cdots\ (n \geq 0)$, which represents the times of CR-TS's recovery. From each of these points, for any S_n, service can be taken as a restart. In other words, the probability of S_n occurring is 1. According to the regenerative process theory, we call time points $S_1,\ S_2,\ \cdots S_n$ regeneration points (see Section 3.2). Since the CR-TS's lifetime Z is a regenerative process and the sequence $\{S_1,\ S_2,\ \cdots S_n\}$ constructs the lifetime of the service, each point has no effect on the following one.

In order to describe the lifetime of the CR-TS more precisely, the following random variables are defined. A stochastic process Z is a set of random variables $\{X(n),\ n \in \mathcal{N}\}$. Let $X(n)$ be the random variable representing the time which elapses from the time of the nth regeneration point to the $(n+1)$th one. For the simplicity of presentation, we

use X instead of $X(n)$ in the following parts. In our case X_a is the random variable representing the time which elapses from the time that the CR-TS starts at one regeneration point to the time the CR-TS fails and X_c is the random variable representing the time which elapses from the time that the CR-TS fails to the time of the next regeneration point. X_a and X_c are independent of each other.

Lemma 3.4.1. *In steady state, the time between each two consecutive CR-TS's recovery time points is one period of the crash-recovery service's lifetime and the CR-TS is also an alternating renewal process.*

Proof. Clearly, from Assumption 3.4.1, the recovery times can be regarded as regeneration points. In steady state, the regeneration point can be regarded as restart, i.e. the behavior of the CR-TS between two consecutive regeneration points is exactly the same. Thus the time between each two consecutive regeneration points is one period of CR-TS's lifetime.

From the above definitions and the definition of an alternating renewal process, we can easily get $X = X_a + X_c$, and $\{(X_a,\ X_c)\}$ is an alternating renewal process. Hence a CR-TS is an alternating renewal process. □

Now, we can conclude that in order to design a failure detector for the CR-TS, which depends on CR-TS's behavior, we only need to observe one period of this CR-TS since all of the other periods are the same as this one.

3.4.2 Dependability of a Crash-Recovery Service

In order to characterize the steady state behaviors of the CR-TS, general dependability measure metrics are adopted, such as *reliability*, *availability* and *consistency* introduced in Section 2.2.5, captured by MTTF, $\frac{\text{MTTF}}{\text{MTBF}}$ and MTTR, respectively. In this section we will show how these three service dependability metrics apply to a CR-TS.

Definition 3.4.1. *Let Z_a be a sequence of $\{X_a^1,\ X_a^2,\ \ldots,\ X_a^i,\ \ldots\}$, where X_a^i represents the length of time that Z is in the* Alive *state; let Z_c be a sequence of $\{X_c^1, X_c^2, \ldots, X_c^i, \ldots\}$, where X_c^i represents the length of time that Z is in the* Crash *state, $i = 1,\ 2,\ \ldots$.*

Proposition 3.4.1. *For a crash-recovery service, the following relations hold:*

$$\begin{aligned} E[Z_a] &= \text{MTTF}; \\ E[Z_c] &= \text{MTTR}; \\ E[Z] &= E[Z_a] + E[Z_c] \\ &= \text{MTTF} + \text{MTTR} \\ &= \text{MTBF}. \end{aligned} \tag{3.4.1}$$

Proof. This is immediate from the definitions of MTTF, MTTR and MTBF in Section 2.2.5. □

Definition 3.4.2. *Let $F(x)$ be the probability distribution of X, $f(x)$ the probability density function of $F(x)$. Let $F_a(x)$ be the probability distribution of X_a, $f_a(x)$ the probability density function of $F_a(x)$. Let $F_c(x)$ be the probability distribution of X_c, $f_c(x)$ the probability density function of $F_c(x)$.*

$$\begin{aligned} E[Z] = \text{MTBF} &= \int_0^{+\infty} x dF(x) = \int_0^{+\infty} x f(x) dx; \\ E[Z_a] = \text{MTTF} &= \int_0^{+\infty} x dF_a(x) = \int_0^{+\infty} x f_a(x) dx; \\ E[Z_c] = \text{MTTR} &= \int_0^{+\infty} x dF_c(x) = \int_0^{+\infty} x f_c(x) dx. \end{aligned}$$

Definition 3.4.3. *Let P_a be the probability that a CR-TS is in the* Alive *state at time t ($t \geq 0$). Let P_c be the probability that a CR-TS is in the* Crash *state at time t ($t \geq 0$).*

Lemma 3.4.2. *If $0 < E(X) < +\infty$, then*

$$\lim_{t \to +\infty} P_a = \frac{\text{MTTF}}{\text{MTBF}} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}};$$

$$\lim_{t \to +\infty} P_c = \frac{\text{MTTR}}{\text{MTBF}} = \frac{\text{MTTR}}{\text{MTTF} + \text{MTTR}}.$$

Proof. When $0 < E(X) < +\infty$, then

$$\lim_{t \to +\infty} P_a = \frac{E[Z_a]}{E[Z]} = \frac{E[Z_a]}{E[Z_a] + E[Z_c]} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} = \frac{\text{MTTF}}{\text{MTBF}};$$

$$\lim_{t\to+\infty} P_c = \frac{E[Z_c]}{E[Z]} = \frac{E[Z_c]}{E[Z_a]+E[Z_c]} = \frac{\text{MTTR}}{\text{MTTF}+\text{MTTR}} = \frac{\text{MTTR}}{\text{MTBF}}.$$

Hence the proof is completed. □

From the above parts, we can see that the dependability of a CR-TS can be quantitatively measured by MTTF, MTTR and MTBF. The reliability property of the CR-TS, which captures how long a CR-TS can persist without a failure, can be measured by MTTF. The availability of the CR-TS, which captures the probability that a CR-TS is alive at an arbitrary time, can be measured by $P_a = \frac{\text{MTTF}}{\text{MTBF}}$. The consistency of the CR-TS, which captures the speed of a CR-TS's recovery, can be measured by MTTR. We will analyze the relationship between the CR-TS's dependability and the QoS of the FDS later in this chapter.

3.5 Probabilistic Message behaviors and QoS of Communication

3.5.1 Failure Detection Communication Channels

In order to measure the communication between the FDS and the target service (TS) quantitatively, we define the communication path between the FDS and the TS as a channel. Each communication component pair holds one or more virtual one-way message source-to-destination channels. The messages can only flow from the source component to the destination component. Fig. 3.3 shows the channel communication between a failure detection pair in push and pull mode.

From Fig. 3.3, we can easily see that: for *push-style* failure detection, the communication channel is a one-way channel; for *pull-style*, there will be two communication channels, which forms a return route. We also associate some properties with each channel assigned with a message type (e.g., heartbeat messages through a channel from a TS to a FDS) :

(a) Push (b) Pull

Figure 3.3: Push Mode and Pull Mode

- QoS of the communication: this is a measurement of how good the communication is and the main communication QoS metrics can be captured by the following properties:
  - Message delay (D): the time interval from the message sending time to the message arrival time.
  - Message loss rate (X_R): the number of lost messages per time unit.
  - Message loss proportion (X_P): the number of lost messages out of the total number of sent messages during some time duration.
  - Message loss-length (the number of loss X_L): the consecutive message loss number. (We will define these properties as random variables shortly.)
- Algorithm: the failure detection algorithm adopted for this monitoring channel, which will indicate how the liveness messages communicate (e.g., push: heartbeat, pull: query, pull: reply). The algorithm has the following properties:
  - Interval: the time duration between two consecutive liveness messages' generation. This is a measurement of how frequently the communication pair talk with each other.
  - Timeout: how long the algorithm needs to wait for the arrival of the liveness message.
  - Protocol: transmission protocol to carry the message (e.g., TCP, UDP).

3.5.2 Probabilistic Measurements of Message Transmission

In addition to the channels defined above, we take the message transmission behavior as probabilistic. We describe the message delay or loss as probabilistic message-based behaviors associated with the communication channel.

Definition 3.5.1. *Let D be a random variable representing the time which elapses from the time a message is sent until the time it arrives the destination; let X_R be a random variable representing the number of message losses per unit time; let X_P be a random variable representing the proportion of the lost messages; let X_L be a random variable representing the number of consecutive messages lost.*

For *push-style* monitoring, let us assume that each heartbeat message's transmission is independent, thus each heartbeat message transmission can be regarded as a Bernoulli trial. So D in Definition 3.5.1 represents the message delay of communication channel's QoS properties; X_R represents the message loss rate; X_P represents the message loss proportion of a communication channel and $E(X_P)$ approximates p_L; X_L represents the consecutive message loss number of a communication channel.

For the above quantities, there are some internal connections between them. For D and p_L, it is easy to understand that the probability of a message loss can be regarded as the probability that a message is delayed infinitely. Thus $p_L = Pr(D = +\infty)$.

For X_P and X_R, assume that within time duration T ($T >> 0$), on average, N_L liveness messages are lost during the transmission and the inter-sending time between liveness messages is η. Therefore $E(X_P) = \frac{N_L}{T/\eta}$ (almost surely (a.s.)) and $E(X_R) = \frac{N_L}{T}$ (a.s.). Since the $E(X_P) = p_L$, thus we obtain:

$$E(X_R) = \frac{E(X_P)}{\eta} = \frac{p_L}{\eta} \ (a.s.) \tag{3.5.2}$$

For p_L and X_L, the following lemma holds:

Lemma 3.5.1. *If each heartbeat message's transmission and loss behavior is independent, then the probability that x ($x \geq 1$) consecutive messages are lost is*

$$Pr(X_L = x) = p_L^x \cdot (1 - p_L).$$

Proof. For x consecutive messages $m_i,\ m_{i+1},\ \cdots,\ m_{i+x-1}$, suppose the event of message m_i being lost is M_i and the event of the message being not lost is $\overline{M_i}$. Then the probability that x of them are lost is

$$Pr(X_L = x) = Pr(M_i \bigcap M_{i+1} \bigcap \cdots M_{i+x-1} \bigcap \overline{M}_{i+x}).$$

Since each messages transmission is independent, then the message loss of each message can be regarded as a Bernoulli trial. Thus

$$Pr(X_L = x) = Pr(M_i) \cdot Pr(M_{i+1}) \cdots\ Pr(M_{i+x-1}) \cdot (1 - Pr(M_{i+x})),$$

since the probability of message m_i loss $Pr(M_i) = p_L$, therefore

$$Pr(X_L = x) = p_L^x \cdot (1 - p_L).$$

Hence the proof is completed. □

From the descriptions in this section, we have shown that the interaction between the FDS and the CR-TS can be regarded as channel-based communication. This channel-based communication can be described by the QoS of the communication, the adopted failure detection algorithm and the adopted communication protocol, each of which has some associated properties. In the following sections we analyze how the FDS monitors the CR-TS and how the FDS can be configured based on this channel-based communication.

3.6 QoS of the Crash-Recovery FDS

3.6.1 System Model

We consider a distributed system model with two services: one FDS and one CR-TS, distributed over a wide-area network. The FDS and the CR-TS are connected by an unreliable communication channel (see Section 3.5.1). Liveness messages are transmitted through the channel. The communication channel does not create or duplicate

liveness messages, but the messages might be lost or delayed indefinitely during transmission[2]. The CR-TS can fail by crashing but can be repaired and restart to run again after some repair time, which behaves as a *crash-recovery* model. The drift of the local clocks of the FDS and the CR-TS are small enough to be ignored and their local clocks are synchronized (this can be guaranteed by some time synchronization service such as the Network Time Protocol used in [36]) to be regarded as a clock synchronized system[3]. The failure detection algorithm we adopted is the NFD-S algorithm proposed in [24]. (A brief introduction of NFD-S algorithm is available in Section 2.4 and the algorithm pseudo-code is available in Appendix A.)

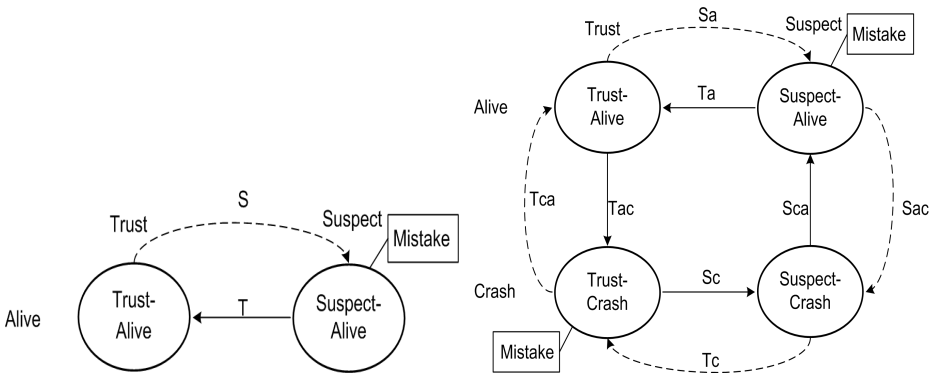
3.6.2 Modeling a Push-Style Crash-Recovery FDS

Previous work on the QoS of failure detection, such as [24, 51, 68, 81], is based on a *fail-free* or *crash-stop* assumption. The failure detector (FDS) in [24] has a set of suspicion levels $\mathcal{S}_s := \{Trust, Suspect\}$. The FDS can either trust or suspect a CR-TS's liveness. Thus for a *fail-free* run, a service only has one state: *Alive*. The state space of a FDS is $\mathcal{S}_f := \{Trust\text{-}Alive, Suspect\text{-}Alive\}$, and the event space of a FDS $\mathcal{F} := \{S\text{-}transition, T\text{-}transition\}$ (Fig. 3.4(a)). For a *fail-free* run, the QoS metrics of a FDS can be measured quite straightforwardly. The average time spent in the *Trust* state, is the mean length of the good period $E(T_G)$; the average time spent in the *Suspect* state, is the mean time of the mistake duration $E(T_M)$; the average time between two consecutive transfers to the *Suspect* state (two consecutive *S-transition*s) is the mean time of the mistake recurrence $E(T_{MR})$.

However, precisely speaking, the state space of a FDS $\mathcal{S}_c := \mathcal{S} \times \mathcal{S}_s$. Therefore, for a CR-TS with failures, the state space of its FDS increases because the service has more than one state (see Fig 3.4(b)). If the suspicion level is more than two, then $\mathcal{S}_c$ will increase as well. The QoS metrics of a FDS are no longer as simple as for *fail-free* runs.

[2]This channel-based message transmission is the same as the probabilistic network model in [24].

[3]We also consider the message delay estimation for the system with unsynchronized clocks in Chapter 4.



(a) Fail-Free Transition (b) Crash-Recovery Transition

Figure 3.4: Crash-Recovery State Space

In order precisely to present the accuracy of the FDS, let $\mathcal{S}_A \in \mathcal{S}_a$ represent the accuracy of the FDS's current output value, where $\mathcal{S}_a := \{Accurate,\ Mistake\}$. Here *Accurate* means the current FDS's output value presents the CR-TS's current state accurately. *Mistake* means the FDS's output value presents the CR-TS's current state inaccurately. Let $\mathcal{S}_{CR\text{-}TS} \in \mathcal{S}$ represent the current state of the CR-TS, where $\mathcal{S} := \{Alive,\ Crash\}$. $\mathcal{S}_{FDS\text{-}O} \in \mathcal{S}_s$ represents the current output value of the FDS, where $\mathcal{S}_s := \{Trust,\ Suspect\}$ is the state space of the suspicion levels of the FDS. Then the logical relationship between the FDS's output, accuracy and the CR-TS's current state is showed in Table 3.1.

$\mathcal{S}_{FDS\text{-}O}$	$\mathcal{S}_{CR\text{-}TS}$	$\mathcal{S}_A$
$Trust(True)$	$Alive(True)$	$Accurate(True)$
$Suspect(False)$	$Alive(True)$	$Mistake(Fasle)$
$Trust(True)$	$Crash(False)$	$Mistake(False)$
$Suspect(False)$	$Crash(False)$	$Accurate(True)$

Table 3.1: The FDS's Accuracy Expression

In Table 3.1, if each of *Trust*, *Alive* and *Accurate* is regarded as True and each of *Suspect*, *Crash*, and *Mistake* as False, the FDS's current accuracy can be derived from

the following deduction:

$$\mathcal{S}_{FDS\text{-}O}\ (XNOR)\ \mathcal{S}_{CR\text{-}TS} \Rightarrow \mathcal{S}_A.$$

The above deduction can be simply reasoned from the fact that if the value of $\mathcal{S}_{FDS\text{-}O}$ and $\mathcal{S}_{CR\text{-}TS}$ are the same then the value of $\mathcal{S}_A$ is True, because the FDS's output value presents the CR-TS's state accurately. If the value of $\mathcal{S}_{FDS\text{-}O}$ and $\mathcal{S}_{CR\text{-}TS}$ are different then the value of $\mathcal{S}_A$ will be False. Thus the value of $\mathcal{S}_A$ is the result of an Exclusive-NOR operation between the value of $\mathcal{S}_{FDS\text{-}O}$ and $\mathcal{S}_{CR\text{-}TS}$.

For a *fail-free* run (MTTF $\rightarrow +\infty$) or a *crash-stop* run (MTTR $\rightarrow +\infty$), the CR-TS's current state $\mathcal{S}_{CR\text{-}TS}$ will always be *Alive* (for the time up to the crash) and it is easy to deduce the FDS's accuracy $\mathcal{S}_A$ directly from the FDS's current state $\mathcal{S}_{FDS\text{-}O}$. However, for a *crash-recovery* run, since the CR-TS could fail or recover at arbitrary time, $\mathcal{S}_A$ cannot be deduced solely by using $\mathcal{S}_{FDS\text{-}O}$. Therefore, measuring the accuracy of a FDS for a CR-TS is more complex.

Compared with a *fail-free* or *crash-stop* run, there are more mistake types in a *crash-recovery* run. In previous work, such as [24, 9, 36, 51, 68, 71, 81], only the mistakes caused by the message transmission behaviors (message delay and loss) are considered. But in a *crash-recovery* run, a mistake starts when the CR-TS's and FDS's states become different. Thus there are also mistakes caused by the CR-TS's crash (see T_F in Fig. 3.1 or T_M^3 in Fig. 3.5(c)) and recovery (see Fig. 3.5(d)) due to the delayed detection of such occurred events. Therefore, there are more types of mistake in a *crash-recovery* run. Fig. 3.5 shows the four types of mistake which could occur within a *crash-recovery* run. T_M^1 in Fig 3.5(a) represents the type of mistake caused by a message delay. T_M^2 in Fig 3.5(b) represents the type of mistake caused by a message loss. T_M^3 in Fig 3.5(c) represents the type of mistake caused by CR-TS's crash, while the FDS still trusts the CR-TS. T_M^4 in Fig 3.5(d) represents the type of mistake caused by CR-TS's recovery, while the FDS still suspects the CR-TS. From Fig. 3.5 we can see that a mistake can be caused by different reasons. A message loss or delay will result in a *Suspect-Alive* mistake of the FDS (see Fig. 3.4(b)). A crash failure will result in a *Trust-Crash* mistake. Therefore the state of the FDS will vary due to a mistake caused by different reasons. In addition, a recovery event will result in a *Suspect-Alive* mistake

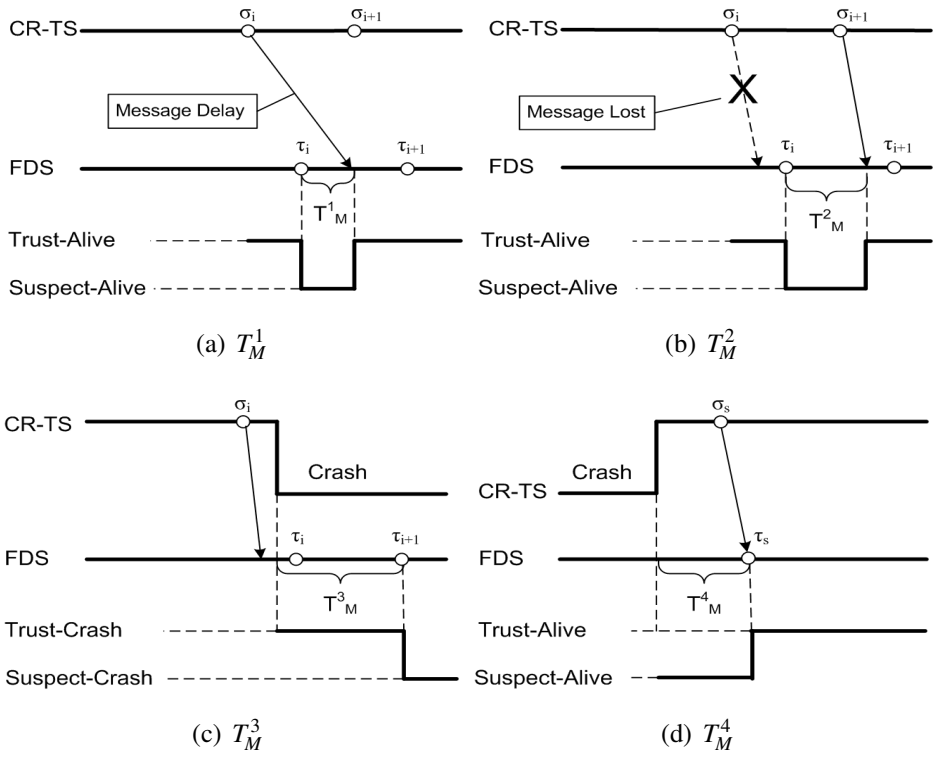


Figure 3.5: The Analysis of Possible T_M in a Crash-Recovery Run

as well, however compared with the mistake caused by the message loss or delay, the FDS is forced to a mistaken state from a different correct state. From Fig. 3.4(b), we can see that a recovery event will trigger a state transition of the FDS from the *Suspect-Crash* state to the *Suspect-Alive* State, but a message delay or loss will trigger a state transition of the FDS from the *Trust-Alive* state to the *Suspect-Alive* State. Moreover, a mistake caused by different reasons will result in a different FDS parameters reconfiguration plan. For instance, the best way for the FDS to tolerate more message losses or a longer message delay is to increase the timeout duration; the best way for the FDS to minimize the mistake duration caused by a crash event is to decrease the timeout duration and the best way to minimize the mistake duration caused by a recovery event is to increase the liveness message sending frequency. Thus we can see that an inaccurate mistake type identification might reduce the QoS of a FDS and should be avoided.

From the above analysis, we can see that due to the increasing mistake types in a *crash-recovery* run, the definition of the QoS metrics in [24] using transitions are not valid in

a *crash-recovery* run. Thus we redefine them as below:

- **Detection time** (T_D): the elapsed time from when the monitored target crashes until the failure detector correctly suspects the monitored target.
- **Mistake recurrence time** (T_{MR}): the time between the occurrence of two consecutive mistakes.
- **Mistake duration** (T_M): the time to correct a mistaken suspect or trust.
- **Average mistake rate** (λ_M): the average number of mistakes per unit time.
- **Good period duration** (T_G): the duration for which the failure detector maintains the correct state information.
- **Query accuracy probability** (P_A): the probability that the state information from the failure detector is correct at an arbitrary time.
- **Forward good period duration** (T_{FG}): the duration for which at a random time, the failure detector has the correct information of the monitored target to the next mistake occurs.

The above QoS metrics can measure some QoS aspects of a failure detector in a *crash-recovery* run. However, they still cannot measure how fast a recovery can be detected, the proportion of the detected failures over the occurred failures (*completeness*), etc. In the following section, we will extend some QoS metrics to measure the recovery detection speed and the *completeness* of a failure detector.

3.6.3 QoS Metrics Extension for the Crash-Recovery FDS

For a *crash-recovery* FDS, in addition to the QoS metrics introduced in [24] (T_D, T_M, T_{MR}, P_A, etc, which will be represented as the basic QoS metrics in later parts), we propose some additional QoS metrics as follows:

First, in order to measure the speed that a FDS can discover a recovery of the CR-TS, a new measurement: **the recovery detection time** (T_{DR}), which represents the time that elapses from the CR-TS's recovery time (a *R-transition* occurs) to the time when the

FDS discovers the recovery of the CR-TS, is adopted. More precisely, T_{DR} is a random variable representing the time that elapses from the time that the CR-TS recovers to the time when the FDS detects the CR-TS's recovery. If there is no recovery detected then $T_{DR} = +\infty$.

Then, as in a *crash-recovery* run there is no eventual behavior of a CR-TS, a fast recovery could make a failure remain undetectable by a FDS. Under such circumstance, the *completeness* property of a failure detector defined in [22] cannot be satisfied any more (see Section 2.3.1). In order to reflect this situation, we refine the definition of the *completeness* as follows:

- *Strong completeness*: every crash failure of a recoverable process will be detected.
- *Weak completeness*: a proportion of crash failures of a recoverable process will be detected, satisfying a specified requirement.

Therefore, in order to measure the *completeness* property of a *crash-recovery* failure detector, we propose some new QoS metrics:

- **The detected failure proportion** (R_{DF}): the ratio of the detected crashes over the occurred crashes ($0 \leq R_{DF} \leq 1$). More precisely R_{DF} is a random variable representing a number between zero and one. When no crash failure is detected, $R_{DF} = 0$. When all of the occurred crashes are detected, $R_{DF} = 1$. The *strong completeness* property of a FDS's requires that $E(R_{DF}) = 1$. The *weak completeness* property requires $E(R_{DF}) \geq R^L_{DF}$ (R^L_{DF} is the required lower bound of the detected failure proportion and $0 \leq R^L_{DF} \leq 1$).
- **The detected recovery proportion** (R_{DR}): the ratio between the detected recoveries over the occurred recoveries ($0 \leq R_{DR} \leq 1$). More precisely R_{DF} is a random variable. When no recovery is detected, $R_{DR} = 0$. When all of the occurred recoveries are detected, $R_{DR} = 1$.
- **The detected failures recurrence time** (T_{DFR}): the time duration between two detected failures ($X \leq T_{DFR} \leq +\infty$). More precisely, T_{DFR} is a random variable. When no crash is detected, $E(T_{DFR}) = +\infty$. When all of the occurred failures

are detected $E(T_{DFR}) = E(X) =$ MTBF (see Fig. 3.6).

- **The detected recoveries recurrence time** (T_{DRR}): the time duration between two detected recoveries ($X \leq T_{DRR} \leq +\infty$). More precisely, T_{DRR} is a random variable. When no recovery is detected, $E(T_{DRR}) = +\infty$. When all of the occurred recoveries are detected, $E(T_{DRR}) = E(X) =$ MTBF (see Fig. 3.6).

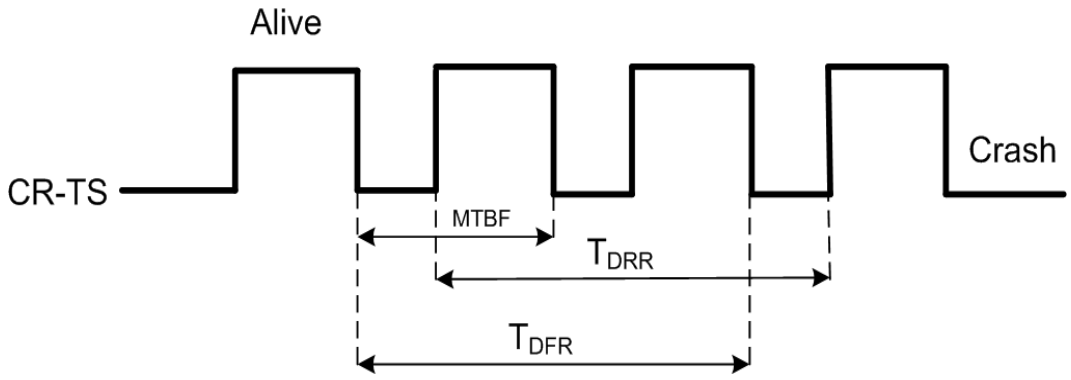


Figure 3.6: Extended QoS Metrics

3.6.4 Relations between the Extended QoS Metrics

From the definition of the extended QoS metrics presented in Section 3.6.3, we can easily find that there are some internal connections between the extended QoS metrics. In Theorem 3.6.1 below, we state the relationships between the R_{DF}, R_{DR}, T_{DFR}, T_{DRR}.[4]

Theorem 3.6.1. *For a crash-recovery FDS, the following results hold:*

$$E(R_{DF}) = E(R_{DR}). \tag{3.6.3}$$

$$E(R_{DF}) = \frac{\text{MTBF}}{E(T_{DFR})},\ E(R_{DR}) = \frac{\text{MTBF}}{E(T_{DRR})}. \tag{3.6.4}$$

$$E(T_{DFR}) = E(T_{DRR}). \tag{3.6.5}$$

[4]We only consider steady state behavior of the FDS and CR-TS pair in a long-run, which can be assumed as an infinite observation duration.

Proof. The equation (3.6.3) can be easily derived from the fact that if a crash is detected, even if the recovery next to the occurred crash is not detected, for a *crash-recovery* CR-TS, eventually, there will be a detected recovery. Thus, $E(R_{DF}) = E(R_{DR})$.

For the equation (3.6.4), let us assume T ($T \gg$ MTBF) to be an observation duration of the *crash-recovery* failure detection pair. Thus, within time duration T, the number of the occurred crashes and recoveries approaches to $\lfloor \frac{T}{\text{MTBF}} \rfloor$; the number of the detected crashes and recoveries approaches to $\lfloor \frac{T}{E(T_{DFR})} \rfloor$ and $\lfloor \frac{T}{E(T_{DRR})} \rfloor$ respectively. Therefore, from the definitions of R_{DF} and R_{DR}, $E(R_{DF}) = \frac{\text{MTBF}}{E(T_{DFR})}$ and $E(R_{DR}) = \frac{\text{MTBF}}{E(T_{DRR})}$ can be easily obtained. Then, the equation (3.6.5) can be obtained directly from (3.6.3) and (3.6.4). Hence, the proof is completed. □

From Theorem 3.6.1, we can conclude that using only one of R_{DF}, R_{DR}, T_{DFR} and T_{DRR} is enough to measure the *completeness* property of a *crash-recovery* FDS. For simplicity, we choose R_{DF} as the primary metric, since it is straightforward. Overall, the QoS for a *crash-recovery* FDS can be captured by P_A, T_M, T_{MR}, T_D, T_{DR}, R_{DF}. In next section, we will analyze the QoS bounds of the NFD-S algorithm in a *crash-recovery* run by adopting the proposed basic and extended QoS metrics.

3.6.5 An QoS Estimate of the NFD-S Algorithm in a Crash-Recovery Run

In a *crash-recovery* run, the state of a CR-TS can switch between *Alive* and *Crash*. As we discussed in Section 3.4.1, there is a sequence of regeneration points for the CR-TS, each of which is the recovery time of the CR-TS. In the following we will treat these as also regeneration points of the system consisting of the failure detection pair. This is an approximation made for pragmatic reasons but it can be justified as follows. In order to study the steady state behavior of a CR-TS throughout its lifetime, we only need to observe the time period between two consecutive recovery time of the CR-TS. Fig. 3.7 shows the relationship between a FDS and a CR-TS on the interval $t \in [t_0, t_3)$, where both t_0 and t_3 are regeneration points. Obviously, the mean time between t_0 and t_3 is the MTBF. We split $[t_0, t_3)$ into $[t_0, t_1)$, $[t_1, t_2)$, $[t_2, t_3)$,

- t_1 is the time when the FDS detects the recovery of the CR-TS from the *Crash* state to the *Alive* state;
- t_2 is the time when the service crashes;
- σ_s is the first liveness message sending time after a recovery;
- σ_f is the sending time of the last liveness message before a crash;
- σ_i is the sending time of a liveness message between σ_s and σ_f;
- η is the liveness sending interval; τ_s is the first decision time after recovery[5];
- τ_b is the last decision time before crash;
- τ_f is the freshness point according to σ_f;
- T_{DR} is the time to detect a recovery.

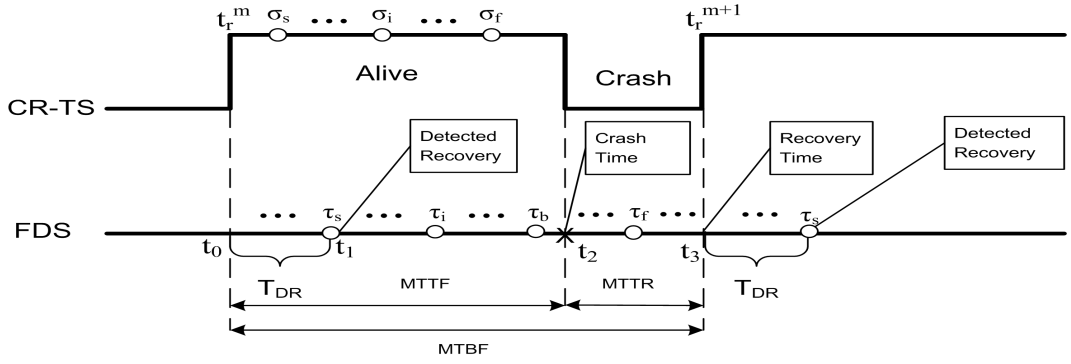


Figure 3.7: The Analysis of the Crash-Recovery NFD-S Algorithm

Let t_r be a recovery time of the current MTBF period (the recovery time is t_0 and t_3 in Fig. 3.7). The following definitions are based on the NFD-S algorithm as extensions of the Definition 1 in [24], both of which are introduced briefly in Section 2.4.

Definition 3.6.1. *For the fail-free duration* $[t_1, t_2)$ *within each MTBF period:*

1. *k: for any* $i \geq 1$*, let k be the smallest integer such that, for all* $j \geq i+k$*,* m_j *is sent at or after time* τ_i.[6]

[5]The first liveness message receiving time.

[6]k is assumed to be independent of i approximately. In fact, in a *crash-recovery* run, k is not com-

2. *For any* $i \geq 1$, *let* $p_j^i(x)$ *be the probability that the FDS does not receive just the* $(i+j)$*th message* m_{i+j} *by time* $\tau_i + x$, *for every* $j \geq 0$ *and every* $x \geq 0$*; let* $p_0^i = p_0^i(0)$.

3. *For any* $i \geq 2$, *let* q_0^i *be the probability that the FDS receives message* m_{i-1} *before time* τ_i.

4. *For any* $i \geq 1$, *let* $u^i(x)$ *be the probability that the FDS suspects the CR-TS at time* $\tau_i + x$, *for every* $x \in [0,\ \eta)$.

5. p_s^i*: for any* $i \geq 2$, *let* p_s^i *be the probability that an S-transition occurs at time* τ_i.

According to the QoS analysis of the NFD-S algorithm in Proposition 3 in [24], we now analyze the basic QoS metrics of the NFD-S algorithm in a *crash-recovery* run and show the following relations hold:

Proposition 3.6.1.

1. $k = \lceil timeout/\eta \rceil$.

2. *for all* $j \geq 0$ *and for all* $x \geq 0$,

$$p_j^i(x) = (p_L + (1-p_L) \cdot Pr(D > timeout + x - j\eta)) \cdot Pr(X_a > \tau_i - t_r + x).$$

3. $q_0^i = (1-p_L) \cdot Pr(D < timeout + \eta) \cdot Pr(X_a > \tau_i - t_r)$.

4. *For all* $x \in [0, \eta)$,

$$u^i(x) = \prod_{j=0}^{k} p_j^i(x).$$

5. $p_s^i = q_0^i \cdot u^i(0)$.

Proof. 1. Since the message sending time of m_i is $\tau_i - timeout$, we know that the sending time of m_j is $\tau_i - timeout + (j-i)\eta$. As the message j is sent after i, then we know that $\tau_i - timeout + (j-i)\eta \geq \tau_i$. Therefore, from the definition of k, we can conclude that $timeout \leq k \cdot \eta \Rightarrow k \geq \frac{timeout}{\eta}$. Since k is the smallest integer number, thus $k = \lceil \frac{timeout}{\eta} \rceil$.

pletely independent of i. However, due to the fact that if the CR-TS will have a reasonable *alive* duration, k will be almost independent of i except the last a few messages before the CR-TS crashes.

2. Within $[t_1,\ t_2)$, $p_j^i(x)$ is the probability that the CR-TS has not crashed before $\tau_i + x$ since the last recovery ($Pr(X_a > \tau_i - t_r + x)$) but the message m_{i+j} might be either lost (p_L) or delayed more than $\tau_i + x - \sigma_j = timeout + x - j\eta$ (represented by $Pr(D > timeout + x - j\eta)$). Thus

$$p_j^i(x) = (p_L + (1 - p_L) \cdot Pr(D > timeout + x - j\eta)) \cdot Pr(X_a > \tau_i - t_r + x).$$

3. Within $[t_1, t_2)$, q_0^i is the probability that the CR-TS has not crashed before $\tau_i + x$ since the last recovery ($Pr(X_a > \tau_i - t_r + x)$), and that the message m_{i-1} is not lost and delayed less than $timeout + \eta$, which can be represented as $(1 - p_L) \cdot Pr(D < timeout + \eta)$. Thus

$$q_0^i = (1 - p_L) \cdot Pr(D < timeout + \eta) \cdot Pr(X_a > \tau_i - t_r).$$

4. Within $[t_1, t_2)$, $u^i(x)$ is the probability that the CR-TS has not crashed before $\tau_i + x$ since the last recovery ($Pr(X_a > \tau_i - t_r + x)$) and that the FDS does not receive any message m_j within $i \le j \le i + k$ by time $\tau_i + x$. Since we assume that each message's transmission is independent, then $u^i(x)$ can be represented by the product of the probability that each message m_j is lost. Therefore, by the definition of $p_j^i(x)$,

$$u^i(x) = \prod_{j=0}^{k} p_j^i(x).$$

5. Within $[t_1, t_2)$, p_s^i is the probability that a mistaken suspect happens when the CR-TS has not crashed. Then p_s^i is the probability that the message m_{i-1} is received by the FDS before time τ_i (the FDS is in the *Trust-Alive* state before time τ_i), and that no message m_j, with $j \ge i$, is received by the FDS by time τ_i (a *Suspect* transition will occur by time τ_i) while the CR-TS has not crashed before τ_i ($Pr(X_a > \tau_i - t_r)$), then according to the definitions of q_0^i and $u^i(x)$,

$$p_s^i = q_0^i \cdot u^i(0).$$

Hence the proof is completed. □

In the following lemma, we are trying to estimate the upper bound of $E(N)$ in a given *fail-free* duration. Let N be a random variable that represents the number of mistakes within a *fail-free* duration $t - t_s$, where $t_s \in [0,\ t)$ and $t < +\infty$. Then we can estimate $E(N)$ as below:

Lemma 3.6.1. *For a FDS in a fail-free run, the mean number of mistakes of the FDS, $E(N)$, is given by:*

$$E(N) \leq (\lfloor \frac{t-t_s}{\eta} \rfloor + 1) \cdot p_s^i.$$

Proof. Obviously, in a *fail-free* run, a mistake can only happen at the freshness point (when an expected message does not arrive). Thus, from Table 3.1, we can conclude that the probability of the mistake occurring when the CR-TS is not crashed, is equal to the probability that an *S-transition* happens. Up to the time t, there will be no more than $\lfloor \frac{t-t_s}{\eta} \rfloor + 1$ freshness points. If each message's transmission is independent, then $E(N) \leq \sum_{n=i}^{i+\lfloor \frac{t-t_s}{\eta} \rfloor + 1} p_s^i$, where i is the sequence number of the freshness point, $i \geq 1$. Thus, within a *fail-free* duration, the mean number of mistakes can be estimated by using:

$$E(N) \leq \sum_{n=i}^{i+\lfloor \frac{t-t_s}{\eta} \rfloor + 1} p_s^i = (\lfloor \frac{t-t_s}{\eta} \rfloor + 1) \cdot p_s^i.$$

□

In the following lemma, the upper and the lower bounds of the average mistake numbers within a *crash-recovery* duration are to be estimated. Let N^* be a random variable that represents the number of mistakes within a *crash-recovery* duration. Then $E(N^*)$ can be estimated as below:

Lemma 3.6.2. *For a crash-recovery FDS, the mean number of mistakes within one crash-recovery period, $E(N^*)$, is given by:*

$$E(N^*) \geq (\lfloor \frac{\text{MTTF} - E(T_{DR})}{\eta} \rfloor + 1) \cdot p_s^i + 1$$

and

$$E(N^*) \leq (\lfloor \frac{\text{MTTF} - E(T_{DR})}{\eta} \rfloor + 1) \cdot p_s^i + \lceil \frac{E(D)}{\eta} \rceil + 2.$$

Proof. Within the interval $[t_0,\ t_3)$, we can distinguish three distinct phases (see Fig. 3.7). While the mean duration of $[t_0,\ t_3)$ is MTBF; the mean duration of $t_0,\ t_1)$ is $E(T_{DR})$; the mean duration of $[t_1,\ t_2)$ is MTTF $- E(T_{DR})$ and the mean duration of $[t_2,\ t_3)$ is MTTR. The mistakes occur in different ways in each of these phases. Therefore in order to estimate the total expected number of mistakes throughout $[t_0,\ t_3)$, we estimate the upper and lower bounds of expected number of mistakes in each of the phases respectively, then we can estimate $E(N^*)$ in the following four steps:

Step I: $[t_0,\ t_1)$ starts from the time of the CR-TS's recovery to the time when the FDS detects the CR-TS's recovery. Obviously, in this interval, there will be at most one mistake which might happen—the FDS does not detect the recovery of the CR-TS. When $X_c \geq \eta + timeout$ (the analysis of the relationship of the crash duration, η and *timeout* is given in the proof of Lemma 3.6.9), a crash failure will be detected before the CR-TS's next recovery, which means the FDS will be in the *Suspect* state before the CR-TS's recovery. Then if the CR-TS recovers from the *Crash* state, a mistake will happen. Since we start observing the CR-TS from a regeneration point (in steady state), we can say this mistake must happen under the condition $X_c \geq \eta + timeout$. If $X_c < \eta + timeout$ (we assume MTTF $>> \eta + timeout$ to avoid infinite *crash-recovery* loops within detection time), then at the time of the CR-TS's recovery, the FDS might still be in a false *Trust* state (see Table 3.1). When the CR-TS recovers, the FDS's false state fortunately becomes a correct state without detecting an occurred failure. Thus, there is no mistake happening within $[t_0,\ t_1)$, i.e. the number of mistakes within $[t_0,\ t_1)$ is zero. Let $N^{'}$ be a random variable that represents the number of mistakes within $[t_0,\ t_1)$, then the upper bound and the lower bound of $E(N^{'})$ can be simply estimated as:

$$\begin{aligned} &\text{IF} \quad X_c > \eta + timeout,\ E(N^{'}) \leq 1; \\ &\text{ELSE IF} \quad X_c < \eta + timeout,\ E(N^{'}) \geq 0. \end{aligned} \tag{3.6.6}$$

Step II: Let $N^{''}$ be a random variable that represents the number of mistakes on $[t_1,\ t_2)$. $[t_1,\ t_2)$ is a *fail-free* duration from the time when the FDS detects the CR-TS's recovery until the time when the CR-TS crashes. On $[t_1,\ t_2)$, the average duration of $t_2 - t_1$ is MTTF $- E(T_{DR})$, then the average number of freshness point is less than $\lfloor \frac{\text{MTTF} - E(T_{DR})}{\eta} \rfloor + 1$. In order to simplify the estimation procedure, we use Lemma 3.6.1,

then the mean number of mistakes, which are occurring within the *fail-free* duration $[t_1,\ t_2)$ in a *crash-recovery* run, can be estimated by:

$$E(N^{''}) \leq \sum_{n=i}^{i+\lfloor \frac{t_2-t_1}{\eta} \rfloor+1} p_s^{'} = (\lfloor \frac{\text{MTTF} - E(T_{DR})}{\eta} \rfloor + 1) \cdot p_s^i. \tag{3.6.7}$$

Step III: $[t_2,\ t_3)$ starts from the time of the CR-TS's crash until the time of the CR-TS's next recovery. In this period, if $X_c > \eta + timeout$, on average there are at most $\lceil \frac{E(D)}{\eta} \rceil$ liveness messages which have been sent out successfully before the CR-TS crashes but which might still be on the way to the FDS (e.g., $E(D) = 0.2$, $\eta = 1$ then there might be one message still in transmission even after the CR-TS crashes). These might generate *false positive* mistakes. If $X_c < \eta + timeout$, in the minimum, there can be no mistake within $[t_2,\ t_3)$. Thus, the number of mistakes that might happen within $[t_2,\ t_3)$ is discussed in the following circumstances:

- Minimum: the FDS has already been in the *Suspect* state and there is no heartbeat message received throughout $[t_2,\ t_3)$, thus no mistake will occur.
- Maximum: if the FDS is in the *Trust* state, when the CR-TS crashes, then there is one mistake caused by the crash (see Fig 3.5(c)). In addition, from the discussion above, we know that there are $\lceil \frac{E(D)}{\eta} \rceil$ heartbeat messages still in transmission. If all of them are not lost and for each of them the delay is $timeout < D < timeout + \eta$ (e.g., message m_i arrives between (τ_i, τ_{i+1}), consequently $\lceil \frac{E(D)}{\eta} \rceil$ number of *S-transition*s will occur at each decision time (τ_i) and a *T-transition* will occur when the message is received). Thus there will be at most $\lceil \frac{E(D)}{\eta} \rceil + 1$ mistakes that will occur within $[t_2,\ t_3)$.

Let $N^{'''}$ be a random variable which represents the number of mistakes within $[t_2,\ t_3)$. Therefore, the mean number of mistakes within $[t_2,\ t_3)$ can be estimated as below:

$$0 \leq E(N^{'''}) \leq \lceil \frac{E(D)}{\eta} \rceil + 1. \tag{3.6.8}$$

In general, $E(D)$ is smaller than η, as in the simulations in [9, 24, 36, 81]. Thus for $E(D) \leq \eta$, on average there might be one message still on its way to the FDS. Then at most there will be two mistakes which could happen within $[t_2,\ t_3)$:

- One mistake:
  - If $X_c < \eta + timeout$, the FDS could possibly trust the CR-TS throughout $[t_2,\ t_3)$, then there will be one mistake occurring within $[t_2,\ t_3)$.
  - If $X_c > \eta + timeout$, let σ_f ($f \geq 1$) be a heartbeat message sending time (see Fig. 3.7). In the following circumstances, there will be only one mistake occurring within $[t_2,\ t_3)$.
    - The FDS is in the *Trust* state and the CR-TS crashes after σ_f but before σ_{f+1} and the FDS receives the heartbeat message m_f before τ_f.
    - The FDS is in the *Trust* state and the CR-TS crashes after σ_f but before σ_{f+1} and the heartbeat message m_f is lost or arrives after τ_{f+1}.
    - The FDS is in the *Suspect* state and CR-TS crashes after σ_f but before σ_{f+1} and the FDS receives the heartbeat message m_f before τ_{f+1}.
- Two mistakes:

  When the FDS is in the *Trust* state and the CR-TS crashes after σ_f but before $\sigma_{f+1}$and the FDS receives the m_f after τ_f but before τ_{f+1} (MTTR $>>\eta$), there will be two mistakes within $[t_2,\ t_3)$. The first mistake is caused by the state change of the CR-TS and the second one is caused by the delayed heartbeat message m_f arriving after τ_f but before τ_{f+1}. For the NFD-S algorithm, in a *fail-free* run, if a heartbeat message's delay $D > timeout + \eta$, it will be discarded and the FDS will suspect the target service permanently. This is because $Pr(D > timeout + \eta)$ should be small enough to satisfy the QoS requirement of the T_{MR}^L. However there is a possibility that a heartbeat message's delay is bigger than $timeout + \eta$, and it might be received by the FDS. In a *crash-recovery* run, there is no permanent suspect and termination. When a fresh liveness message is received the FDS will trust the CR-TS again.

Step IV: Combining inequalities 3.6.6, 3.6.7 and 3.6.8, the upper bound of the mean

number of mistakes that will happen within $[t_0,\ t_3)$ is given by:

$$\begin{aligned} E(N^*) &\leq 1 && [t_0,\ t_1) \\ &\quad + (\lfloor \frac{\text{MTTF} - E(T_{DR})}{\eta} \rfloor + 1) \cdot p_s^i && [t_1,\ t_2) \\ &\quad + \lceil \frac{E(D)}{\eta} \rceil + 1 && [t_2,\ t_3) \\ &\leq (\lfloor \frac{\text{MTTF} - E(T_{DR})}{\eta} \rfloor + 1) \cdot p_s^i + \lceil \frac{E(D)}{\eta} \rceil + 2. \end{aligned}$$

For the lower bound of the mean number of mistakes that will happen within $[t_0,\ t_3)$, obviously, if there is no mistake within $[t_0,\ t_1)$, there must be at least one mistake occurring within $[t_2,\ t_3)$. If there is no mistake within $[t_2,\ t_3)$, there must be at least one mistake occurring within $[t_0,\ t_1)$. Thus,

$$\begin{aligned} E(N^*) &\geq (1 \quad \text{or} \quad 0) && [t_0,\ t_1) \\ &\quad + (\lfloor \frac{\text{MTTF} - E(T_{DR})}{\eta} \rfloor + 1) \cdot p_s^i && [t_1,\ t_2) \\ &\quad + (0 \quad \text{or} \quad 1) && [t_2,\ t_3) \\ &\geq (\lfloor \frac{\text{MTTF} - E(T_{DR})}{\eta} \rfloor + 1) \cdot p_s^i + 1. \end{aligned}$$

The proof is completed.

□

Lemma 3.6.3. *For a crash-recovery service FDS,*

$$E(T_{MR}) \geq \frac{\text{MTBF}}{(\lfloor \frac{\text{MTTF} - E(T_{DR})}{\eta} \rfloor + 1) \cdot p_s^i + \lceil \frac{E(D)}{\eta} \rceil + 2}.$$

Proof. In a *crash-recovery* run, the CR-TS reaching steady state means that the CR-TS crashes and recovers every MTBF on average, rather than that it never fails. Therefore, the mean time of mistake recurrence ($E(T_{MR})$) is the observation duration (MTBF) over the mean number of mistakes that happen within every MTBF.

$$\begin{aligned} E(T_{MR}) = \frac{E(X)}{E(N^*)} &\geq \frac{\text{MTBF}}{\left(1 + (\lfloor \frac{\text{MTTF} - E(T_{DR})}{\eta} \rfloor + 1) \cdot p_s^i + \lceil \frac{E(D)}{\eta} \rceil + 1\right)} \\ &\geq \frac{\text{MTBF}}{(\lfloor \frac{\text{MTTF} - E(T_{DR})}{\eta} \rfloor + 1) \cdot p_s^i + \lceil \frac{E(D)}{\eta} \rceil + 2}. \end{aligned} \tag{3.6.9}$$

□

Lemma 3.6.4. *For a crash-recovery FDS,*

$$E(T_{MR}) \leq \text{MTBF}.$$

Proof. For a *crash-recovery* FDS, as the CR-TS crashes or recovers, the probability that the output of the FDS changes simultaneously with the state change of the CR-TS is zero. If the FDS always trusts the CR-TS or always suspects the CR-TS, then there will be at least one mistake when the CR-TS crashes or recovers, thus $E(T_{MR}) \leq$ MTBF. □

Lemma 3.6.5. *For a crash-recovery FDS, if a crash failure is detected before the CR-TS's recovery ($X_c > \eta + timeout$), then*

$$E(T_{MR}) \leq \frac{\text{MTBF}}{2}.$$

Proof. According to Lemma 3.6.3, if there is no mistake caused by message delays or losses, then at least there will be one mistake caused by the CR-TS's crash and one mistake caused by the CR-TS's recovery when the failure is detected before the CR-TS's recovery. If the FDS has already suspected the CR-TS before it crashes, it means there is at least one mistake caused by message delay or loss. If the FDS has already trusted the CR-TS before it recovers, it means there is at least one *false positive* mistake caused by delayed message arrival. So there will be at least two mistakes for each MTBF. Thus if $X_c > \eta + timeout$, $E(T_{MR}) \leq \frac{\text{MTBF}}{2}$. The proof is completed. □

Lemma 3.6.6. *For a crash-recovery FDS, within each* MTBF*, the average mistake duration is:*

$$E(T_M) \leq \frac{E(T_{DR}) + \frac{1}{\eta} \cdot \int_0^{\eta} u^i(x)dx \cdot (\text{MTTF} - E(T_{DR})) + E(T_D)}{(\lfloor \frac{\text{MTTF} - E(T_{DR})}{\eta} \rfloor + 1) \cdot p_s^i + 1}.$$

Proof. In order to estimate the total duration of mistakes within the time interval $[t_0,\ t_3)$, we first estimate the average mistake durations within $[t_0,\ t_1)$, $[t_1,\ t_2)$, $[t_2,\ t_3)$ respectively. Let $T_M^{'}$, $T_M^{''}$, $T_M^{'''}$ be the mistake durations within $[t_0,\ t_1)$, $[t_1,\ t_2)$, $[t_2,\ t_3)$ respectively. Therefore within one MTBF period (see Fig. 3.7), $E(T_M^{'})$, $E(T_M^{''})$ and $E(T_M^{'''})$ can be derived as follows:

- Estimate $E(T_M^{'})$: On $[t_0,\ t_1)$, obviously, the average mistake duration is equal to the average time to detect the recovery of the CR-TS. Since there is only one mistake within this duration, the average total mistake duration within $[t_0,\ t_1)$ is equal to $E(T_{DR})$ as well. Therefore $E(T_M^{'})$ can be estimated as below:

$$E(T_M^{'}) = E(T_{DR}).$$

- Estimate $E(T_M^{''})$: On $[t_1,\ t_2)$, since the duration of $[t_1,\ t_2)$ is the time which elapses from when a recovery of the CR-TS is detected to the time of the CR-TS's next crash, then the average duration of $[t_1,\ t_2)$ is $\text{MTTF} - E(T_{DR})$ and $[t_1,\ t_2)$ can be regarded as a *fail-free* duration. Thus $E(T_M^{''})$ can be estimated by using $P_A = 1 - E(T_M)/E(T_{MR})$. Since the probability of the accuracy of the FDS can be calculated by using $P_A^{''} = 1 - \frac{1}{\eta} \cdot \int_0^{\eta} u^i(x)dx$ (see Lemma 15 in [24]). On $[t_1,\ t_2)$, if there will be $E(N'')$ mistakes on average, $E(T_{MR})$ within $[t_1,\ t_2)$ can be estimated by using $E(T_{MR}^{''}) = \frac{\text{MTTF} - E(T_{DR})}{E(N^{''})}$. Therefore combined with inequality 3.6.7, $E(T_M^{''})$ can be estimated as follows:

$$\begin{aligned} E(T_M^{''}) &= (1 - P_A^{''}) \times E(T_{MR}^{''}) \\ &= \frac{\frac{\text{MTTF} - E(T_{DR})}{\eta} \cdot \int_0^{\eta} u^i(x)dx}{E(N^{''})} \\ &= \frac{\frac{\text{MTTF} - E(T_{DR})}{\eta} \cdot \int_0^{\eta} u^i(x)dx}{(\lfloor \frac{\text{MTTF} - E(T_{DR})}{\eta} \rfloor + 1) \cdot p_s^i} \end{aligned}$$

Then the total duration of a FDS in the mistake state within $[t_1,\ t_2)$ can be estimated by the following method. Let $T_M^1,\ T_M^2,\ \cdots,\ T_M^i,\ \cdots,\ T_M^n$ be the mistake durations that occur within $[t_1,\ t_2)$ and assume that the total number of mistakes within $[t_1,\ t_2)$, n, is finite. Thus $\sum_{i=1}^{n} T_M^i$ can be calculated by using the following equation:

$$\begin{aligned} \sum_{i=1}^{n} T_M^i &= E(T_M^{''}) \times E(N^{''}) \\ &= \frac{\text{MTTF} - E(T_{DR})}{\eta} \cdot \int_0^{\eta} u^i(x)dx. \end{aligned} \tag{3.6.10}$$

- Estimate $E(T_M^{'''})$: On $[t_2,\ t_3)$, if the occurred failure is detectable ($X_c > \eta + timeout$), the average total mistake duration within $[t_2,\ t_3)$ should be less than the average

detection time $E(T_D)$, because after a failure is detected, the FDS terminates a mistaken trust. If the occurred failure is undetectable ($X_c < \eta + timeout$), the average total mistake duration within $[t_2,\ t_3)$ should be less than MTTR. This is because after the CR-TS's recovery, the mistaken trust will become a correct trust. In addition, when the occurred failure is undetectable, we will know that X_c is less than $\eta + timeout$, then MTTR ($E(X_c)$) will be less than $E(T_D)$ ($\eta + timeout$). Therefore the total mistake duration and the average mistake duration should be less than $E(T_D)$.

$$E(T_M''') \leq E(T_D).$$

From the above analysis, we know that the total mistake duration on $[t_0,\ t_1)$ is $E(T_{DR})$; the total mistake duration on $[t_1,\ t_2)$ is $\sum_{i=1}^{n} T_M^i = \frac{\text{MTTF}-E(T_{DR})}{\eta} \cdot \int_0^{\eta} u^i(x)dx$; the total mistake duration on $[t_2,\ t_3)$ is less than $E(T_D)$. Thus the average mistake duration in a *crash-recovery* run ($E(T_M)$) can be estimated using the inequality

$$E(T_M) \leq \frac{E(T_{DR}) + \sum_{i=1}^{n} T_M^i + E(T_D)}{E(N^*)}.$$

From Lemma 3.6.2, we know that $E(N^*) \geq (\lfloor \frac{\text{MTTF}-E(T_{DR})}{\eta} \rfloor + 1) \cdot p_s^i + 1$. Thus if we substitute $\sum_{i=1}^{n} T_M^i$ by equation 3.6.10 and substitute $E(N^*)$ by the above analysis result, $E(T_M)$ can be estimated as follows:

$$E(T_M) \leq \frac{E(T_{DR}) + \frac{\text{MTTF}-E(T_{DR})}{\eta} \cdot \int_0^{\eta} u^i(x)dx + E(T_D)}{(\lfloor \frac{\text{MTTF}-E(T_{DR})}{\eta} \rfloor + 1) \cdot p_s^i + 1}.$$

Hence the proof is completed. □

Lemma 3.6.7. *For a crash-recovery FDS,*

$$P_A \geq 1 - \frac{E(T_D) + E(T_{DR}) + \frac{\text{MTTF}-E(T_{DR})}{\eta} \cdot \int_0^{\eta} u^i(x)dx}{\text{MTBF}}.$$

Proof. In a *crash-recovery* run, the probability of a FDS's accuracy means the probability that the output of the FDS expresses the correct state of the CR-TS (see Fig. 3.4(b) and Table 3.1). This can be either the FDS in the *Trust* state and the CR-TS in the *Alive* state or the FDS in the *Suspect* state and the CR-TS in the *Crash* state (See

Fig 3.1). Therefore, for each MTBF duration, P_A can be derived from the proportion of the total time duration that the FDS is in *Accurate* state over the total observation time (MTBF).[7] For each MTBF duration, the durations of the FDS in *Mistake* state are known as $E(T_{DR})$ within $[t_0, t_1)$, $\sum_{i=1}^{n} T_M^i$ within $[t_1, t_2)$ and $E(T_D)$ within $[t_2, t_3)$. Thus the probability of accuracy is given by

$$P_A \geq 1 - \frac{E(T_{DR}) + \sum_{i=1}^{n} T_M^i + E(T_D)}{\text{MTBF}}$$

According to equation 3.6.10, P_A can be estimated as follows

$$P_A \geq 1 - \frac{E(T_D) + E(T_{DR}) + \frac{\text{MTTF} - E(T_{DR})}{\eta} \cdot \int_0^{\eta} u^i(x)dx}{\text{MTBF}}.$$

The proof is completed.

□

Lemma 3.6.8. *For a crash-recovery FDS, if the crash is detectable and the liveness message restarts simultaneously when the CR-TS recovers, then*

$$E(T_{DR}) = E(D) + \eta \cdot E(X_L).$$

Proof. Let t_r be the recovery time of the CR-TS and assume that the message m_s is the first liveness message sent by the CR-TS after a recovery and the message m_i is the first successful liveness message, which is received by the FDS, where s and i are the sequence numbers of the messages, $0 \leq s \leq i$. From the definition of X_L, we notice that $i - s = X_L$. Thus, the sending time of the m_i can be estimated by $t_r + \eta E(X_L)$ and the receiving time of the m_i can be estimated by $t_r + \eta \cdot E(X_L) + E(D)$. Thus the mean time of discovering the CR-TS's recovery can be estimated by $\eta \cdot E(X_L) + E(D)$, where $\eta \cdot E(X_L)$ represents the elapsed time from the most recent recovery (t_r) until the sending time of message i. Therefore we obtain $E(T_{DR}) = E(D) + \eta \cdot E(X_L)$, which proves the lemma. □

Lemma 3.6.9. *For a crash-recovery FDS, if the crash is detectable and the liveness message restarts simultaneously when the CR-TS recovers, then*

$$E(R_{DF}) \geq Pr(X_c > \eta + timeout).$$

[7] $P_A = \frac{E(T_G)}{E(T_{MR})}$ is used to estimate P_A in [24], but our test results show that if the monitoring runtime is reasonably long, then both estimation methods of P_A will get very similar results.

Proof. Intuitively, if the CR-TS's crash duration (X_c) is shorter than the failure detection duration ($\eta + timeout$), then the occurred crash failure might not be detectable, since after the CR-TS's recovery, it will restart sending liveness messages. If such messages are received before the current $timeout$ threshold time, then the occurred crash failure will remain undetected.

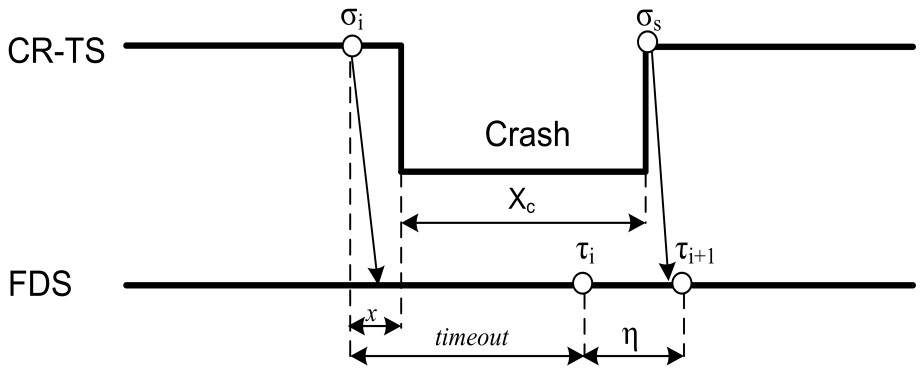


Figure 3.8: The Analysis of R_{DF}

Fig. 3.8 shows the relationship of the crash duration and the failure detection duration and how they can impact R_{DF}. The following is a detailed analysis: assume the CR-TS crashes at time $\sigma_i + x$, where σ_i is the last liveness message sent by the CR-TS before it crashes and x is the time duration elapsed from σ_i to the CR-TS's crash time (for $0 \leq x \leq \eta$). Let τ_i be the freshness point of σ_i, τ_{i+1} be the freshness point next to τ_i, σ_s be a heartbeat sending time after the CR-TS's recovery. From the definition of the NFD-S algorithm, we know that $\tau_i - \sigma_i = timeout$ and $\tau_{i+1} - \tau_i = \eta$. Notice that from Fig. 3.8, it is shown that if $x + X_c < timeout + \eta$, then an occurred crash might not be detectable. Therefore, in order to detect an occurred crash failure, $x + X_c > timeout + \eta$ is needed. In the worst case, the CR-TS crashes just after σ_i ($x = 0$), then we can see that $X_c > timeout + \eta$ is the condition that the failure is detectable. Assuming that there are a number of failures with the duration X_c^1, X_c^2 ..., X_c^i, ..., X_c^n ($1 < i < n < +\infty$). Let N_F be a random variable presenting the number of the occurred failures; let N_{DF} be a random variable presenting the number of the detected failures among the occurred failures. From the definition of R_{DF} in Section 3.6.3, we know that $R_{DF} = \frac{N_{DF}}{N_F}$ and $E(R_{DF}) = E(\frac{N_{DF}}{N_F})$. Since X_cs are random variables which follow a distribution, then the proportion of X_c^i greater than $timeout + \eta$ can be represented by $Pr(X_c > \eta + timeout)$.

From the above analysis, we know that the average proportion (expectation of ratio) of the detected failures ($E(R_{DF})$) in such a situation is equal to the proportion of X_c^i greater than $timeout + \eta$. Therefore we obtain that $E(R_{DF}) \geq Pr(X_c > \eta + timeout)$, which proves the lemma. □

From Lemma 3.6.3-3.6.9, we can conclude that for the NFD-S algorithm, the basic QoS metrics proposed in [24] and the extended QoS metrics are constrained by the following theorem:

Theorem 3.6.2. *The crash-recovery FDS based on the NFD-S algorithm has the following properties:*

$$\text{MTBF} \geq E(T_{MR}) \geq \frac{\text{MTBF}}{(\lfloor \frac{\text{MTTF}-E(T_{DR})}{\eta} \rfloor + 1) \cdot p_s^i + \lceil \frac{E(D)}{\eta} \rceil + 2}. \tag{3.6.11}$$

If $X_c > \eta + timeout$*, then*

$$\frac{\text{MTBF}}{2} \geq E(T_{MR}) \geq \frac{\text{MTBF}}{(\lfloor \frac{\text{MTTF}-E(T_{DR})}{\eta} \rfloor + 1) \cdot p_s^i + \lceil \frac{E(D)}{\eta} \rceil + 2}. \tag{3.6.12}$$

$$P_A \geq 1 - \frac{E(T_D) + E(T_{DR}) + \frac{\text{MTTF}-E(T_{DR})}{\eta} \cdot \int_0^\eta u^i(x)dx}{\text{MTBF}}. \tag{3.6.13}$$

$$E(T_M) \leq \frac{E(T_{DR}) + \frac{\text{MTTF}-E(T_{DR})}{\eta} \cdot \int_0^\eta u^i(x)dx + E(T_D)}{(\lfloor \frac{\text{MTTF}-E(T_{DR})}{\eta} \rfloor + 1) \cdot p_s^i + 1}. \tag{3.6.14}$$

$$E(T_{DR}) = E(D) + \eta \cdot E(X_L). \tag{3.6.15}$$

$$E(R_{DF}) \geq Pr(X_c > \eta + timeout). \tag{3.6.16}$$

Proof. The proof of Theorem 3.6.2 follows immediately from the application of Lemma 3.6.3-Lemma 3.6.9. □

When the monitoring target is *fail-free* or *crash-stop*[8], for the basic QoS metrics in [24], from (3.6.11) to (3.6.14) in Theorem 3.6.2, we can easily deduce that

$$E(T_{MR}) \geq \frac{\eta}{p_s^i}. \tag{3.6.17}$$

[8]The *pre-crash* duration of the *crash-stop* process is a long run.

$$E(T_M) \leq \frac{1}{p_s^i} \cdot \int_0^{\eta} u^i(x)dx \leq \frac{\eta}{q_0^i}. \tag{3.6.18}$$

$$P_A \geq 1 - \frac{1}{\eta} \cdot \int_0^{\eta} u^i(x)dx. \tag{3.6.19}$$

As MTTF $\rightarrow +\infty$, $Pr(X_a > \tau_i + x - t_r^m)$ approaches one. Therefore, p_s^i, $u^i(x)$ and q_0^i in the Definition 3.6.1 are reduced to p_s, $u(x)$ and q_0 in Definition 1 in [24]. Thus $E(T_{MR})$, $E(T_M)$ and P_A are exactly reduced to the QoS analysis results in [24]. We can conclude that in terms of failure detection, a *fail-free* run or a *crash-stop* run with MTTF $\rightarrow +\infty$ is a particular case of a *crash-recovery* run. If the monitored target's MTTF is not sufficiently long and the target is recoverable, then the impact of its dependability should be taken into consideration.

3.7 The Configuration of the NFD-S Algorithm in a Crash-Recovery Run

For the NFD-S algorithm in a *crash-recovery* run, the assumption that the sequence numbers of the heartbeat messages are continually increasing after every recovery of the CR-TS is needed to ensure that the NFD-S algorithm is still valid after each recovery. However, without persistent storage to snapshot the runtime information frequently, when a crash failure occurs, all of the current runtime information might be lost. Thus continuously increasing the heartbeat sequence number cannot be guaranteed in such a situation. Since for the NFD-S algorithm, the local clocks of the FDS and the CR-TS are synchronized, we can use the comparison of the sending time of each heartbeat message instead of the comparison of the heartbeat sequence number in the NFD-S algorithm. Then, for a *crash-recovery* FDS, if the QoS requirements of the FDS are given, the configuration procedure is illustrated in Fig. 3.9.

Initially, we can assume that the QoS of message communication is perfect (e.g., $p_L = 0$, $E(D)$ is small and $E(X_L) = 0$), and the CR-TS is *fail-free*. As the monitoring procedure continues, the estimation of the QoS of message communication and the dependability metrics of the CR-TS will become more and more accurate. Thus the

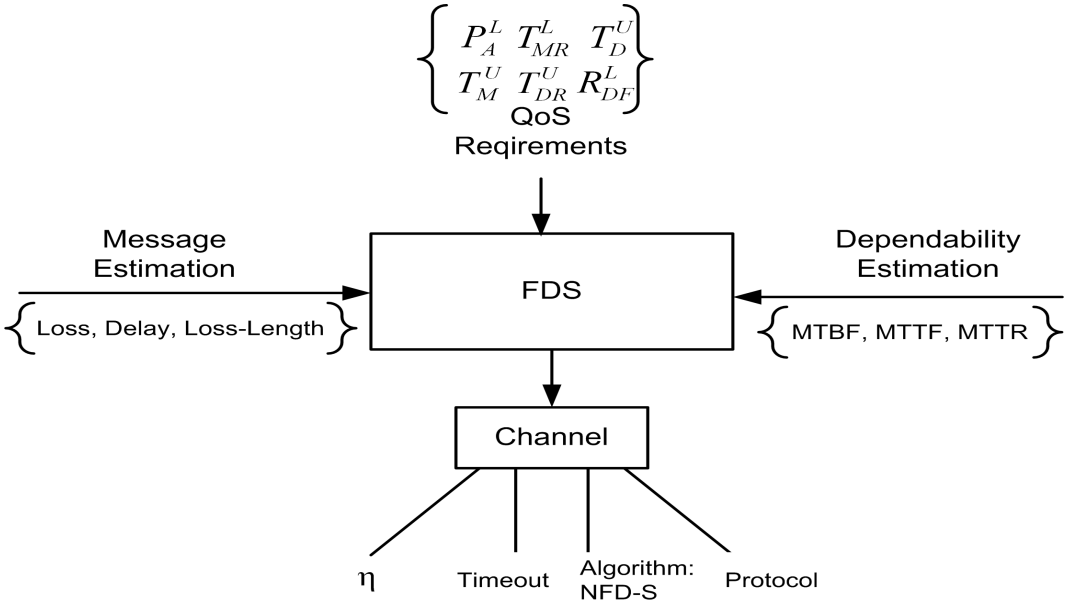


Figure 3.9: The Extended NFD-S Algorithm Configuration in a Crash-Recovery Run

FDS can be reconfigured to adapt to the change of input parameters, which can help estimate better η and $timeout$. Then for given QoS requirements, expressed as bounds, the following inequalities need to be satisfied:

$$\begin{aligned} &T_D \le T_D^U,\ E(T_{MR}) \ge T_{MR}^L,\ P_A \ge P_A^L, \\ &E(T_M) \le T_M^U,\ E(T_{DR}) \le T_{DR}^U,\ E(R_{DF}) \ge R_{DF}^L. \end{aligned} \tag{3.7.20}$$

From Theorem 3.6.2, we can estimate the parameters (η and $timeout$) of the NFD-S algorithm according to the following inequalities:

$$\eta + timeout \le T_D^U,\ \eta > 0. \tag{3.7.21}$$

$$\frac{\text{MTBF}}{(\lfloor \frac{\text{MTTF}-E(T_{DR})}{\eta} \rfloor + 1)\cdot p_s^i + \lceil \frac{E(D)}{\eta} \rceil + 2} \ge T_{MR}^L. \tag{3.7.22}$$

$$1 - \frac{E(T_D) + E(T_{DR}) + \frac{\text{MTTF}-E(T_{DR})}{\eta} \cdot \int_0^\eta u^i(x)dx}{\text{MTBF}} \ge P_A^L. \tag{3.7.23}$$

$$\frac{E(T_{DR}) + \frac{\text{MTTF}-E(T_{DR})}{\eta} \cdot \int_0^\eta u^i(x)dx + E(T_D)}{(\lfloor \frac{\text{MTTF}-E(T_{DR})}{\eta} \rfloor + 1)\cdot p_s^i + 1} \le T_M^U. \tag{3.7.24}$$

$$E(D)+\eta E(X_L) \leq T_{DR}^U. \tag{3.7.25}$$

$$Pr(X_c > \eta + timeout) \geq R_{DF}^L. \tag{3.7.26}$$

Then the configuration of the NFD-S algorithm becomes the problem to find the largest η satisfying inequalities (3.7.22)-(3.7.25) and if such η exists, find the largest $timeout$ that satisfies $\eta + timeout \leq T_D^U$ and $Pr(X_c > \eta + timeout) \geq R_{DF}^L$. The configuration procedure can be done in the following steps:

Step I If $T_{MR}^L <$ MTBF, continue; else the QoS of the FDS cannot be achieved.

Step II Find the largest η that satisfies the inequalities (3.7.22)-(3.7.25), otherwise cannot find an appropriate η (QoS cannot be achieved).

Step III If $\eta > 0$, find the largest $timeout \leq T_D^U - \eta$ and $Pr(X_c > \eta + timeout) \geq R_{DF}^L$.

From the above steps, the estimation of η and $timeout$ for a *crash-recovery* FDS based on the NFD-S algorithm amounts to finding a numerical solution for the inequalities (3.7.21)-(3.7.26). This can be done using binary search similarly to [24]. But the estimation of the input parameters of the configuration becomes more difficult because parameters, such as $E(X_L)$, MTTF, MTTR etc., are needed for such a FDS. We will introduce input parameter estimation shortly in Chapter 4. Note that for this configuration procedure, choosing a different message transmission protocol (e.g., TCP, UDP) can achieve different QoS for message communication. Thus, this new configuration can be more adaptive to the message transmission. For example, if the message loss probability or message delay is high for a certain protocol, then the FDS can switch to a more reliable protocol to achieve a better QoS without increasing the communication frequency or the $timeout$ length.

3.8 Discussion

In previous sections, we introduced how to estimate the QoS bounds for a *crash-recovery* FDS based on the NFD-S algorithm. However, there are several facts which

need to be taken into consideration.

In reality, MTTF and MTTR are non-deterministic values, governed by random distributions. The proportion of detected failures is dependent on the probability distribution of X_c, the length of η and $timeout$. If it is required to detect most failures before recovery, in practice $\eta + timeout$ (the failure detection time T_D) should be much smaller than MTTR. For example, for exponentially distributed X_c, if $\eta + timeout = \text{MTTR}$, then $E(R_{DF}) \geq 36.8\%$. If $\eta + timeout = \frac{\text{MTTR}}{2}$, then $E(R_{DF}) \geq 60.7\%$. If $\eta + timeout = \frac{\text{MTTR}}{10}$, $E(R_{DF}) \geq 90.5\%$.

Theorem 3.6.2 gives the bounds of $E(T_{MR})$ and $E(T_M)$. However, from Fig. 3.7 we can see that the characteristics of $E(T_{MR})$ and $E(T_M)$ in the durations of $[t_0,\ t_1)$ ($E(T'_{MR})$, $E(T'_M)$), $[t_1,\ t_2)$ ($E(T''_{MR})$, $E(T''_M)$), $[t_2,\ t_3)$ ($E(T'''_{MR})$, $E(T'''_M)$) are quite different. Estimating η and $timeout$ using the mean of the dependability measurements might not satisfy the QoS requirement all the time. A stricter bound can be achieved by using the maximum value in the set $\{E(T'_M),\ E(T''_M),\ E(T'''_M)\}$, which must be smaller than T_M^U and the minimum value in $\{E(T'_{MR}),\ E(T''_{MR}),\ E(T'''_{MR})\}$, which must be larger than T_{MR}^L.

For T_M, $E(T'_M)$ is $E(T_{DR})$; $E(T'''_M)$ is less than $E(T_D)$; the $E(T''_M)$, which is the *fail-free* duration can be estimated by using the inequality 3.6.18. Thus $E(T''_M) \leq \frac{1}{p_s^i} \cdot \int_0^\eta u^i(x)dx \leq \frac{\eta}{q_0^i}$. Then the mistake duration within each MTBF can be as follows:

$$\max\left(E(T_{DR}),\ \frac{\eta}{q_0^i},\ E(T_D)\right) \leq T_M^U. \tag{3.8.27}$$

More strictly, $E(T_{DR})$ can be substituted by the max recovery detection time that has been recorded and $E(T_D)$ can be substituted by T_D^U.

For T_{MR}, the possible mistake recurrence of the FDS is affected by the message delays, losses, the CR-TS's crashes and recoveries. The impact of the CR-TS's crash and recovery is governed by MTTF and MTTR, which mainly occur during $[t_0,\ t_1)$ and $[t_2,\ t_3)$. The impact of message delays and losses on T_{MR} mainly happen within $[t_1,\ t_2)$ represented as $E(T''_{MR})$, which can be estimated using the following inequality:

$$E(T''_{MR}) \geq \frac{\text{MTTF} - E(T_{DR})}{E(N'')}. \tag{3.8.28}$$

According to the inequality (3.6.7) and (3.8.28):

$$E(T_{MR}^{''}) \geq \frac{\text{MTTF} - E(T_{DR})}{(\lfloor \frac{\text{MTTF} - E(T_{DR})}{\eta} \rfloor + 1) \cdot p_s^i}. \tag{3.8.29}$$

When $\text{MTTF} - E(T_{DR}) >> \eta$

$$E(T_{MR}^{''}) \geq \frac{\eta}{p_s^i}. \tag{3.8.30}$$

Therefore, $E(T_{MR})$ can be estimated by using the minimum value in the set $\{\text{MTTF}, E(T_{MR}^{''}), \text{MTTR}\}$. Then the bound estimation of $E(T_{MR})$ can be reduced as follows:

$$\min\left(\text{MTTF}, \frac{\eta}{p_s^i}, \text{MTTR}\right) \geq T_{MR}^L. \tag{3.8.31}$$

Inequality (3.8.31) gives a stricter constraint for the QoS estimation. However, the drawback of this method is obvious. For a highly consistent CR-TS, due to small MTTR, T_{MR} could be too small to satisfy a given QoS requirement. In this situation, using $E(T_{MR})$ instead could be a reasonable solution because the recovery of the CR-TS only happens once per MTBF period. Furthermore, if *timeout* is scaled up or even becomes larger than MTTR, from Theorem 3.6.2 we can know that the $E(T_{MR})$ can increase, but more failures will become undetectable. For such highly consistent CR-TS, some new algorithm is needed to tackle this problem. Thus, in Chapter 4, the recovery detection protocols are presented to discover a failure after the recovery and estimate the recovery time, which can improve the $E(R_{DF})$ without reducing other QoS aspects.

3.9 The Impact of Service Dependability Metrics on the QoS of the FDS

In order to assess the impact of the dependability metrics on the FDS's QoS, some distinct cases are distinguished as follows. When $\text{MTTF} \to +\infty$ (or $\text{MTTF} >> 0$), the CR-TS stays in the *Alive* state for a very long time. This indicates that the CR-TS is highly reliable. In this situation, regardless the value of MTTR, the service seldom

crashes and remains in the *Alive* state for long periods. Such a service can be regarded as a *fail-free* run and the impact of the service dependability is so low that it can be ignored. When MTTF $>>$ MTTR, the CR-TS is highly available. But the CR-TS might not be highly reliable, e.g., MTTF $>>$ MTTR does not indicate MTTF $\rightarrow +\infty$. If MTTF is not large enough then the behavior of the CR-TS will influence the QoS of the FDS. We will discuss this in the following part. When MTTR $\rightarrow 0$, the CR-TS is highly consistent. When MTTR $>> 0$, the CR-TS can be regarded as a *crash-stop* service. When MTTF $>$ MTTR or MTTF $<$ MTTR, but MTTR is not large enough, then the impact of failure and recovery on the QoS of a FDS has to be considered. When MTTF $<<$ MTTR, it means that in most time the CR-TS is in the *Crash* state. In this situation, the CR-TS can rarely send heartbeat messages or answer the query. Such a service is too poor to be useful, so we do not consider this case. In the following sections, we will discuss the impact of the reliability, availability and consistency on each QoS metric respectively.

3.9.1 The Impact on T_M and T_D

Generally, for a *timeout* based push or pull algorithm (see Fig. 3.7), the *timeout* length governs the failure detection speed. This is simply because the FDS makes its decision at the *timeout* point. If the *timeout* time is small, the FDS will make faster, but less accurate, decisions. If *timeout* increases, T_D slows down but the FDS can tolerate more message (heartbeat or query-reply) delays or losses (see Fig. 3.10), which can improve the detection accuracy to some extent. Note that if the service is not *fail-free* or *crash-stop*, continually increasing the *timeout* length, a failure might become undetectable, because its recovery duration could be shorter than T_D. In this situation, $E(T_M)$ will not increase more than the recovery duration.

3.9.2 The Impact on T_{MR}

For a *fail-free* run, mistakes of a FDS only happen when messages are delayed or lost and T_{MR} is governed by such message behaviors. Previous research work [24] showed

that when T_D ($timeout$) increases linearly, T_{MR} increases exponentially (Fig. 12 in [24]). This implies that in a *fail-free* run, if the *timeout* length is increased continually, an arbitrary level of T_{MR} can be achieved. In Fig. 3.10, $P = Pr(Delay > timeout) = p_L + (1 - p_L) \cdot Pr(timeout < Delay < +\infty)$, where p_L is the probability of message loss. Roughly speaking, in a *fail-free* run, when *timeout* increases to $n \times \eta$ (n is a non-negative integer and $n \geq 1$), the FDS can tolerate around n consecutive communication message losses. The mistake recurrence which is caused by message latency or loss decreases $\frac{1}{P^n}$ rapidly. This also makes the T_{MR} increase $\frac{1}{P^n}$ faster in a *fail-free* run. For a *crash-recovery* run, mistakes can also happen when the CR-TS crashes

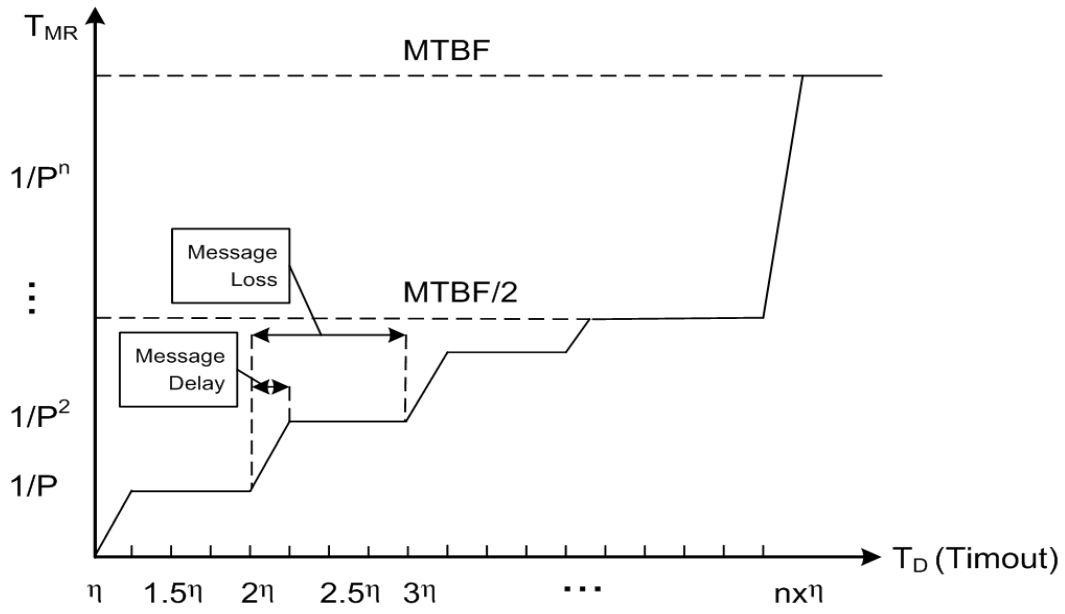


Figure 3.10: The Analysis of $E(T_{MR})$

or recovers (see Fig. 3.4(b)). This is because for any communication-based FDS (in either a synchronous or asynchronous distributed system), the message transmission latency will delay the detection of the CR-TS's state change. This mistake is inevitable even when the prediction is available, because the probability that a predictor predicts the exact time of a state transition of the CR-TS is zero. So a mistake in the FDS will certainly occur when a state transition of the CR-TS happens. This means that the upper bound on T_{MR} is governed by MTTF and MTTR (see inequalities (3.6.11)-(3.6.12) in Theorem 3.6.2). Even if all message delays and losses can be tolerated,

$E(T_{MR})$ cannot increase to an arbitrary level when MTTF is not $+\infty$ and MTTR is not $+\infty$ or 0. Fig. 3.10 shows that if the failure is detectable, $E(T_{MR})$ cannot exceed $\frac{\text{MTBF}}{2}$ since for each MTBF duration there will be two mistakes, caused by the CR-TS's crash and recovery. When the failure is undetectable, mistakes may happen at the CR-TS's crash or recovery time. Then $E(T_{MR})$ cannot exceed MTBF. Fig. 3.10 shows the maximum boundary of $E(T_{MR})$ when *timeout* (T_D) is increased for deterministic MTTF and MTTR. For the CR-TS with the non-deterministic failure and recovery duration, as the *timeout* length increases, the proportion of the detectable failures decreases. For detectable failures, $T_{MR} \leq \frac{\text{MTBF}}{2}$. For undetectable failures, $T_{MR} \leq$ MTBF. Thus after $E(T_{MR})$ reaches $\frac{\text{MTBF}}{2}$, the overall $E(T_{MR})$ approaches MTBF gradually.

3.9.3 The Impact on P_A

Both T_M and T_{MR} are absolute measurements, with strict bounds. But P_A is different. It is the proportion of time that the FDS is not in a mistake state which will depend on the ratio of $E(T_M)$ and $E(T_{MR})$ ($P_A = 1 - \frac{E(T_M)}{E(T_{MR})}$ in [24]). If a service is *fail-free*, from Section 3.9.1 and 3.9.2, we know that when *timeout* is increased, $E(T_M)$ increases linearly and $E(T_{MR})$ increases exponentially. Thus P_A can rapidly approach 1. But in a *crash-recovery* run, when the *timeout* length is increased, both $E(T_M)$ and $E(T_{MR})$ will eventually reach their upper bounds. Generally, as *timeout* increases, less failures will be detected and the mistake caused by the CR-TS's crash (see T_M^3 in Fig. 3.5(c)) will have more impact on $E(T_M)$, thus $E(T_M)$ will approach MTTR[9], since the maximum length of $E(T_M^3)$ is MTTR. When the *timeout* length become more and more larger than $E(X_c)$ (MTTR), more crashes become not detectable. Thus $E(T_M)$ will approach MTTR gradually. For instance, for an exponentially distributed X_c, the probability that a crash is not detected can be calculated by $Pr(X_c < T_D) = 1 - e^{T_D/\text{MTTR}}$ (see proof of Lemma 3.6.9). When $T_D =$ MTTR, the probability that a crash is not detected approaches $Pr(X_c <$ MTTR$)$. As X_c is an exponential distributed random variable, $Pr(X_c <$ MTTR$) = 1 - e^{-1} \approx 0.6321$. When $T_D =$ 2MTTR, $Pr(X_c <$ 2MTTR$) = 1 - e^{-2} \approx 0.8647$. When $T_D =$ 3MTTR, $Pr(X_c <$ 3MTTR$) = 1 - e^{-3} \approx 0.9502$. We can

[9]Under the assumption that the probability of message loss and delay are not very high and MTTR $>> \eta$.

see that as T_D increases, more and more crashes will become undetectable. Therefore, if all crashes become undetectable, $E(T_M)$ will approach MTTR.

The speed of increase of T_{MR} will depend on when T_{MR} reaches $\frac{\text{MTBF}}{2}$. Before this bound is reached, as the *timeout* length increases, T_{MR} can increase exponentially fast, when more message losses can be tolerated. After T_{MR} exceeds $\frac{\text{MTBF}}{2}$, it cannot increase exponentially any more, but can only increase gradually to MTBF. This is because, when T_{MR} does not reach $\frac{\text{MTBF}}{2}$, as the *timeout* length increases, more message delays and losses can be tolerated and less mistakes will occur. Thus, T_{MR} can increase $p_L^{\lfloor \frac{timeout - E(D)}{\eta} \rfloor}$ fast, where *timeout* is the power of p_L (see Fig. 3.10). But when T_{MR} exceeds $\frac{\text{MTBF}}{2}$, even more message delays and losses are tolerated, the mistakes still occur which are caused by the CR-TS's crash and recovery. If a crash is detectable, within each MTBF period there will be at least two mistakes; if a crash is undetectable, within each MTBF duration there will be at least one mistake which is caused by the crash or the recovery (see Lemma 3.6.4 and 3.6.5). As *timeout* increases, more and more crashes become undetectable, therefore, T_{MR} increases gradually, approaching MTBF rather than increasing exponentially fast. Thus, when T_{MR} reaches its upper bound but T_M has not reached its upper bound yet, P_A will decrease with continually increasing *timeout*. When both T_M and T_{MR} reach their upper bound, P_A will approach $\frac{\text{MTTF}}{\text{MTBF}}$, which is equal to the availability of the CR-TS.

3.10 Summary and Conclusions

3.10.1 Summary

In this chapter, we model the channel-based QoS of message transmission. The *crash-recovery* service and its failure detector are modeled as stochastic processes. We redefined the previous proposed QoS metrics for the *crash-recovery* failure detection and extend some new metrics to measure the recovery detection speed and the *completeness* property of a failure detector. In addition to the QoS of message transmission, the dependability of the *crash-recovery* service is considered for the failure detector's

QoS bounds analysis. We show how to configure the FDS to satisfy a given set of requirements in a *crash-recovery* run based on the NFD-S algorithm. We also discussed the impact of the target service's *crash-recovery* behavior on each QoS metric of the failure detector specifically. Our analysis also shows that the QoS analysis in [24] is a particular case of a *crash-recovery* run. Finally, according to the analysis work in Section 3.6 and the discussion in Section 3.9, the QoS of failure detectors is influenced by the dependability of the *crash-recovery* service when the target service is not *fail-free* or *crash-stop*.

3.10.2 Conclusions about Crash-Recovery Failure Detectors

The *crash-recovery* failure detector is an important building block for fault-tolerant systems. This is especially the case for large-scale distributed systems, which have a large number of monitored targets and require the monitoring procedure to be autonomously when the monitored services or processes can be resilient and recover. As far as we know, the previous work focused on the QoS of failure detectors is based on *fail-free* or *crash-stop* model. The corresponding failure detection algorithms only focus on measuring the QoS of message transmission and ignore the dependability of the monitored service. Although there are some studies of *crash-recovery* failure detectors for consensus or group membership problems, they are not from the QoS perspective.

In our model, we have considered QoS metrics of failure detectors under *crash-recovery*, showing that the dependability metrics of the target service should be taken into consideration. When MTTF and MTTR do not approach infinity, in another words, if the target service is not *fail-free* or *crash-stop*, the dependability of the target service will influence the QoS of failure detectors. If an accurate failure detector's parameters are to be estimated, the dependability metrics must be used as inputs as well rather than only considering the impact of the liveness message transmission measurements (see Fig. 3.11).

Note that this *crash-recovery* failure detection model is suitable for failure detection of most software or hardware components that can be repaired to restart and can be more widely adopted by any fault-tolerant systems, which require *crash-recovery* detection.

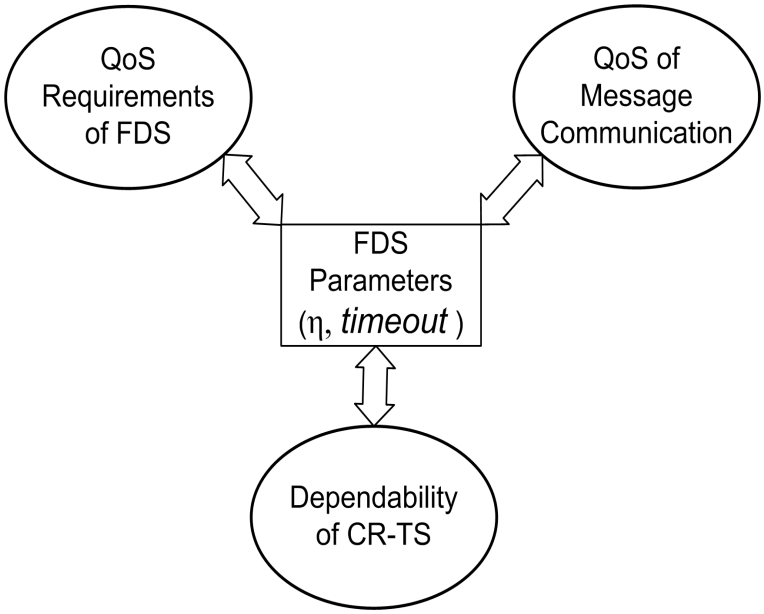


Figure 3.11: The QoS Relationship Between Communication, CR-TS and FDS

Furthermore, our failure detection model focuses on just one failure detection pair. However, in a large distributed system, there might be many services which need to be monitored. For such a system, a failure detector can monitor many targets based on one target per monitoring channel module. The failure detector maintains a list of channel modules' state, each of which presents the liveness status of a monitored target. The parameter calculations and the configuration should be independently based on each channel module. In order to avoid a single point failure of a failure detector, a target service can be monitored by more than one failure detector and a failure detector can be monitored as a target service by other failure detectors as well. Overall, redundancy and diversity can make a failure detection system more robust and dependable. However, the costs of resource consumption for such redundancy will be higher as well.

In next chapter, we will discuss how to estimate the QoS of message transmission and the dependability metrics of the CR-TS. The recovery detection protocols are introduced for estimating the recovery time of the CR-TS, improving the detected failure proportion and improve the adaptation of the FDS in a *crash-recovery* run.

Chapter 4

Parameter Estimation and Recovery Detection Protocol

4.1 Introduction

The results of the previous chapter assume that various parameters characterizing the behavior of the system are available. In this chapter, we explain how such parameters can be estimated. In Section 4.2, we discuss how to estimate the QoS of the liveness message communication in a *crash-recovery* run. Then, we describe how to estimate the dependability metrics of the *crash-recovery* service by using the failure detector itself. In Section 4.3, two types of recovery detection protocols are proposed. The first one is a reliable recovery detection protocol, which requires persistent storage and can detect every recovery. The second one is a lightweight recovery detection protocol, which does not guarantee to detect every recovery but has a lower system overhead.

4.2 QoS of Message Transmission Estimation

In the previous chapter (Section 3.5.1), the message delay, the probability of message loss and message loss-length are defined as the QoS provided by a communication

channel between the FDS and the CR-TS. Some previous work, such as [24, 36, 51, 68, 81], proposes various estimation methods of the message delays and message losses. However all of them are based on the *fail-free* or *crash-stop* assumption and none of them consider how to estimate the message behavior within a *crash-recovery* run. In this section, we discuss the QoS of message communication estimation methods in a *crash-recovery* run and show how the parameter estimation adapts to the *crash-recovery* of the CR-TS.

4.2.1 Estimating E(D) for Each MTBF

Let A_i be the arrival time of the ith heartbeat message sent by the CR-TS according the FDS's local clock. Let σ_i be the sending time of the ith heartbeat message according to the CR-TS's local clock. Let D^i be the delay of the ith heartbeat message, thus $E(D)$ can be estimated as follows:

- For the system with synchronous clocks, since the clocks on both of the FDS and the CR-TS's sides are the same, then $A_i = \sigma_i + D^i$ and $E(D)$ can be estimated by computing the average delay of the received heartbeat messages:

$$\begin{aligned} E(D) &= \frac{1}{n}\sum_{i=1}^{n} D^i \\ &= \frac{1}{n}(\sum_{i=1}^{n}(A_i - \sigma_i)). \end{aligned} \tag{4.2.1}$$

- For the system with unsynchronized clocks, it could be difficult to estimate the average message delay at an arbitrary time. This is because the heartbeat messages start from a recovery time rather than from time zero[1] as in a *crash-stop* run and the heartbeat sending time might be unknown due to the fact that the recovery time recorded by the CR-TS's clock might be invalid at the FDS side. We assume that E(D) from the previous *crash-recovery* period is recorded and available. At the initiation of the failure detection pair, let us assume that the FDS and the CR-TS start at the same time and the heartbeat messages restart

[1] In [24, 36] as in a *crash-stop* run, the authors assume that their failure detectors and the heartbeat generation of the monitored processes start at time zero.

immediately after the CR-TS's recovery. Let s be the starting sequence number of the heartbeat message. Then the sending time of the ith heartbeat message according to the FDS's local clock $\sigma_i + C$ can be represented by $t_r + (i-s)\eta$, where C is a constant representing the drift between the FDS and the CR-TS's clocks[2] and t_r is the recovery time according to the FDS's local clock; the arrival time of the ith heartbeat message A_i can be represented by $\sigma_i + C + D^i$, where D^i is the delay of the ith heartbeat message. Thus for each *crash-recovery* period the delay of the ith heartbeat message can be estimated using the following equations:

$$\begin{aligned} A_i = \sigma_i + C + D^i &= t_r + (i-s)\cdot\eta + D^i; \\ A_i - t_r &= (i-s)\cdot\eta + D^i; \\ D^i &= A_i - t_r - (i-s)\cdot\eta. \end{aligned} \tag{4.2.2}$$

Since the FDS and the CR-TS start simultaneously for the first MTBF period, the first start time t_r according to the FDS's local clock is known. Thus $E(D)$ can be calculated as follows:

$$\begin{aligned} E(D) &= \frac{1}{n}\sum_{i=1}^{n} D^i \\ &= \frac{1}{n}\sum_{i=1}^{n} (A_i - t_r - (i-s)\cdot\eta). \end{aligned} \tag{4.2.3}$$

From the equation (4.2.3), we can get $E(D)$ for the first *crash-recovery* period, thus the previous period's $E(D)$ can be used subsequently. Be aware that this method might result in inaccurate estimation of $E(D)$ after a long time running. One possible compensation solution could be: for each *crash-recovery* period, after the CR-TS's recovery has been detected, the FDS triggers some query messages (using the same message protocol as the heartbeat) to the CR-TS and the CR-TS answers the query. Then, the FDS estimates the roundtrip message delay according to the received answer messages and computes the up-to-date $E(D)$ using its local clock.

[2]We assume that both the FDS and the CR-TS's clocks are accurate.

4.2.2 Estimating p_L for Each MTBF

The parameter p_L can be estimated by using the proportion of messages lost through a communication channel (see Section 3.5.1) based on each MTBF period. For the first MTBF period, initially p_L can be regarded as zero. For the following MTBF periods, initially the previous period's p_L can be used. Within each MTBF period, after the FDS has received some liveness messages, p_L can be recalculated by the following equation:

$$p_L = 1 - \frac{N_r}{N_s}, \tag{4.2.4}$$

where N_r is the number of messages received by the FDS until the estimation time within current MTBF period and N_s is the total number of messages that the CR-TS has sent until the estimation time. N_s can be computed by using $\lfloor \frac{t_{now}-t_r}{\eta} \rfloor + 1$, where t_{now} is the current time and t_r is the recovery time of this period (we leave the details of recovery time estimation to Section 4.3.1 and 4.3.2). Thus,

$$p_L = 1 - \frac{N_r}{\lfloor \frac{t_{now}-t_r}{\eta} \rfloor + 1}. \tag{4.2.5}$$

The above estimation of p_L can be used for the systems with both synchronized clocks and unsynchronized clocks. For the system with unsynchronized clocks, the estimate of the recovery time t_r is less accurate than for the clock synchronized system. But as the number of liveness messages increases, the bias of p_L estimation caused by recovery time estimation becomes negligible.

4.2.3 Estimating Expected Arrival Time For Each MTBF

In a *fail-free* run, Chen *et al.* [24] give an estimation method for the expected message arrival time for the NFD-E algorithm by using

$$EA_{\ell+1} \approx \frac{1}{n}(\sum_{i=1}^{n} A_i - i \cdot \eta) + (\ell+1)\eta,$$

which is based on an exponential message delay assumption. In a *crash-recovery* run, the above method is not valid any more. This is because after a recovery of

the CR-TS, the expected message arrival time needs to be recomputed, depending on the recovery time of the current MTBF period. Therefore, in a *crash-recovery* run, the $(\ell+1)$th heartbeat message's sending time can be estimated by using $t_r + (\ell+1-s)\eta$, and the jth message delay can be computed by using $A_j - t_r - \eta(j-s)$. Thus the $(\ell+1)$th heartbeat message's expected arrival time $EA_{\ell+1}$ can be estimated as below:

$$EA_{\ell+1} \approx t_r + \frac{1}{n}\Big(\sum_{j=1}^{n} A_j - t_r - \eta(j-s)\Big) + (\ell+1-s)\eta, \quad \text{for } s \le j \le (\ell+1), \tag{4.2.6}$$

where j is the heartbeat sequence number, A_j is the arrival time of the jth heartbeat message according to the FDS's local clock, t_r is the recovery time of the current MTBF period and s is the starting heartbeat sequence number of the current MTBF period.

4.2.4 Message Loss-Length Estimation

In addition to $E(D)$ and p_L, the consecutive message loss number X_L is also involved as one QoS aspect of the channel-based communication to capture the bursty message loss behavior. In this section, we propose a basic estimation method to estimate X_L with the independent message transmission assumption.

Lemma 4.2.1. *If each message's transmission and loss behavior is independent, then the mean number of consecutive message losses is* $E(X_L) = \frac{p_L(1-p_L^m)}{1-p_L} - mp_L^{m+1}$*, where* m *is the maximum number of consecutive messages lost and* p_L *is the probability that each message is lost during the transmission.*

Proof. From the definition of expectation and Lemma 3.5.1, we know that

$$\begin{aligned} E(X_L) =& 1 \times p_L \cdot (1-p_L) + 2 \times p_L^2 \cdot (1-p_L) + 3 \times p_L^3 \cdot (1-p_L) + \cdots \\ &+ m \times p_L^m \dot{(}1-p_L) = (1-p_L)\sum_{n=1}^{m} np_L^n, \end{aligned} \tag{4.2.7}$$

where n is the number of consecutive message losses and p_L is the probability that each message is lost. Then,

$$\begin{aligned} p_L \times E(X_L) =& 1 \times p_L^2 \cdot (1-p_L) + 2 \times p_L^3 \cdot (1-p_L) + 3 \times p_L^4 \cdot (1-p_L) + \cdots \\ &+ m \times p_L^{m+1} \cdot (1-p_L). \end{aligned} \tag{4.2.8}$$

From equations 4.2.7 and 4.2.8, we can obtain that

$$E(X_L) - p_L \times E(X_L) = (1-p_L) \times (p_L + p_L^2 + p_L^3 + p_L^4 + \cdots + p_L^m - m p_L^{m+1}).$$

Hence after simple arithmetic manipulation, recognizing that there is a geometry distribution on the right hand side of equation, we get the following equation:

$$E(X_L) = \frac{p_L(1-p_L^m)}{1-p_L} - m p_L^{m+1}.$$

The proof is completed. □

Remark 4.2.1. *When $m \to +\infty$, and $0 < p_L < 1$, then $p_L^m \to 0$, and $m p_L^m \to 0$, we obtain that*

$$E(X_L) = \frac{p_L}{1-p_L}.$$

From the above lemma we see that if each liveness message's transmission is independent, $E(X_L)$ is only related to p_L and it can be computed straightforwardly.

4.2.5 Dependability Metrics Estimation for the Crash-Recovery Service

From the *crash-recovery* service modeling in Section 3.4, we see that there is an intimate relationship between the MTTF, MTTR and MTBF. In order to estimate these dependability metrics of a *crash-recovery* service, we only need to estimate the crash and recovery time of the *crash-recovery* service. Assuming that the clocks between the FDS and the CR-TS's sides are synchronized. Let t_r^1 be the CR-TS's first start time, then for $m \geq 1$, t_r^m represents the mth recovery time; t_{dr}^m represents the mth recovery detection time; t_c^m represents the mth crash time; t_d^m presents the mth crash detection time (see Fig. 4.1). t_r^m can be recorded by the CR-TS after a recovery finished (see

Section 4.3). t_d^m can be recorded by the FDS when a failure is detected, $E(T_D)$ can be estimated by using $\frac{1}{n}\sum_{m=1}^{n}(t_d^m - t_c^m)$ when t_c^m is known. Actually, t_c^m can be estimated by recording the latest successful message sending time σ_l and can be saved in the persistent storage. If a crash event happens uniformly distributed on $[\sigma_l,\ \sigma_l + \eta)$, then after a recovery finished, the average t_c^m can be roughly estimated by $t_c^m = \sigma_l + \frac{\eta}{2}$. Notice that a smaller message inter-sending time unit (η) can result in a more accurate t_c^m estimate. Then the CR-TS's MTBF, MTTF, MTTR and the probability that the CR-TS has not crashed up to time $\tau_i + x$ since its last recovery $Pr(X_a > \tau_i + x - t_r^m)$ can be estimated as follows:

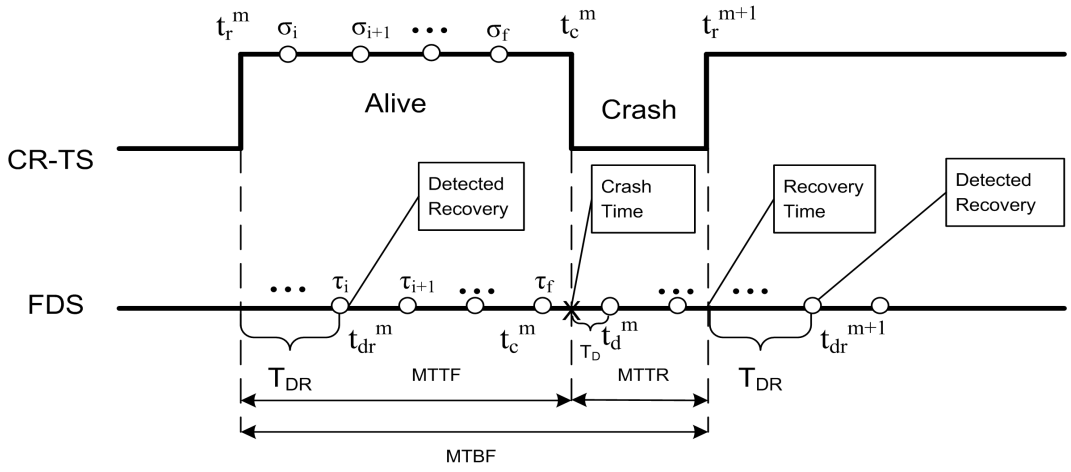


Figure 4.1: Dependability Metrics Estimation

- Estimate MTBF: from the definition of MTBF, we know that MTBF is only related to the CR-TS's recovery times $t_r^m(s)$. These $t_r^m(s)$ can be obtained by adopting the recovery time estimation methods proposed in Section 4.3. Thus MTBF can be estimated as below:

$$\text{MTBF} = E(t_r^{m+1} - t_r^m) = \frac{1}{n}\sum_{m=1}^{n}(t_r^{m+1} - t_r^m). \tag{4.2.9}$$

- Estimate MTTF: MTTF can be estimated by using the recovery time (t_r^m) and

the crash detection time (t_d^m) with the following equation:

$$\begin{aligned} E(t_d^m - t_r^m) &= \text{MTTF} + E(T_D), \text{ then} \\ \text{MTTF} &= E(t_d^m - t_r^m) - E(T_D) \\ &= \frac{1}{n}\sum_{m=1}^{n}(t_d^m - t_r^m) - E(T_D). \end{aligned} \tag{4.2.10}$$

- Estimate MTTR: MTTR can be estimated by using MTBF and MTTF directly for MTTR = MTBF − MTTF or by using t_r^{m+1} and t_d^m. Hence the MTTR can be estimated as below:

$$\begin{aligned} E(t_r^{m+1} - t_d^m) &= \text{MTTR} - E(T_D), \text{ then} \\ \text{MTTR} &= E(t_r^{m+1} - t_d^m) + E(T_D) \\ &= \frac{1}{n}\sum_{m=1}^{n}(t_r^{m+1} - t_d^m) + E(T_D). \end{aligned} \tag{4.2.11}$$

- Estimate $Pr(X_a > \tau_i + x - t_r^m)$: when the probability density function $f_a(x)$ or the probability distribution function $F_a(x)$ of X_a are known, the probability that the CR-TS does not crash until $\tau_i + x$ after its last recovery can be estimated as follows:

$$\begin{aligned} Pr(X_a > \tau_i + x - t_r^m) &= 1 - \int_0^{\tau_i + x - t_r^m} f_a(x)dx \\ &= 1 - F_a(x)|_0^{\tau_i + x - t_r^m}. \end{aligned} \tag{4.2.12}$$

When $x = 0$, we obtain that

$$\begin{aligned} Pr(X_a > \tau_i - t_r^m) &= 1 - \int_0^{\tau_i - t_r^m} f_a(x)dx \\ &= 1 - F_a(x)|_0^{\tau_i - t_r^m}. \end{aligned} \tag{4.2.13}$$

When the probability density function $f_a(x)$ and the probability distribution function $F_a(x)$ of X_a are unknown, an empirical distribution function (EDF) estimation method can be adopted to estimate $f_a(x)$ or $F_a(x)$. In addition, $Pr(X_a > \tau_i + x - t_r^m)$ is used for estimating the probability that a *S-transition* happens on $[t_1,\ t_2)$ (see Proposition 3.6.1), which is used for counting the average number of mistakes on that duration (see Lemma 3.6.1). If we maximize $Pr(X_a > \tau_i + x - t_r^m)$, then a maximal average number of mistakes on $[t_1,\ t_2)$ will

be obtained. Therefore we will get more strict QoS bound estimates for P_A, T_M and T_{MR}. Thus we can adopt $i = 1$ and $x = 0$ to simplify the estimation of $Pr(X_a > \tau_i + x - t_r^m)$. Notice that the above method is only for the strict bound estimation rather than an optimized estimation.

In this section we discussed how to estimate the input parameters of a FDS. The estimation methods are heavily dependent on the recovery time of the CR-TS within each MTBF period. In the next section, we will introduce how to obtain the recovery time for each MTBF period with the proposed recovery detection protocols.

4.3 Recovery Detection and Recovery Time Estimation

In this section, two types of recovery detection protocols are introduced. The first is a reliable protocol which guarantees to detect every recovery of the CR-TS, and the second is a lightweight protocol which does not guarantee to detect every recovery but reduces the system overhead. Both of the two protocols can be adopted to estimate the recovery time of the CR-TS and improve the proportion of detected failures.

4.3.1 A Reliable Recovery Detection Protocol

For the system in which clocks between the FDS side and the CR-TS side are synchronized, the easiest way to detect a recovery and get the recovery time is to send a reliable notification together with the CR-TS's recorded recovery time to the FDS. Then the FDS can detect this recovery and extract this recovery time directly from each received notification message. Thus each recovery can be detected and the recovery time can be recorded. However, if reliable communication is not available, this method will not guarantee to detect every recovery. If a notification message is lost then the recovery time will not be known by the FDS. Consequently, it will result in inaccurate parameter estimation and the failure detection algorithm might not work properly in a *crash-recovery* run. Therefore, in order to detect each occurred recovery and estimate the recovery time accurately, we propose a protocol to track the recovery time by pig-

gybacking some additional information within heartbeat messages and using persistent storage to store the recovery time and the recovery count (see Algorithm 1). For such a recovery detection protocol, the following conditions are needed for recovery time estimation:

- R_m is a positive integer number starting from 0, which represents a count of how many times the CR-TS has recovered.
- The CR-TS has a persistent storage $RECOVERY[N]$ with the length N sufficiently large. Each $RECOVERY[N]$ element can store a set $SET_m := \{R_m, t_r^m\}$, which contains two values: the recovery count R_m and the recovery time t_r^m. Thus, $RECOVERY[N]$ can contain many SET_ms to store multiple recovery time information and initially $RECOVERY[N]$ is empty .
- On each recovery, SET_m is refreshed with current R_m, t_r^m and is stored into $RECOVERY[N]$.

At the CR-TS's side, when the CR-TS finishes its recovery, the CR-TS will add SET_m to $RECOVERY[N]$. Then, all SET_ms in the persistent storage $RECOVERY[N]$ will be piggybacked within the heartbeat message before it is sent to the FDS. When the FDS receives a heartbeat message, it will check whether there is such recovery information available within this heartbeat message. If there are SET_ms within a heartbeat message, the FDS will retrieve and record all R_ms and t_r^ms if they have not been recorded yet. Then the FDS will send an acknowledgement message (m_{ack}) with all of the retrieved R_ms to the CR-TS. If there is no R_m and t_r^m information in the received heartbeat message and the acknowledgement message has already been launched, then the FDS will stop sending the acknowledgement message to the CR-TS. For the CR-TS, if an acknowledgement message is received, according to the R_ms in the acknowledgement message, the CR-TS will retrieve and delete the corresponding SET_ms in $RECOVERY[N]$. If there is no SET_ms in $RECOVERY[N]$, then the CR-TS will stop piggybacking recovery information. By using this protocol, all t_r^ms will be discovered by the FDS, since if the recovery information is not recorded by the FDS, it will remain in the persistent storage for the next piggyback round. Consequently, each occurred failure will be detected. Therefore a *strong completeness* requirement can be satisfied.

For the system with unsynchronized clocks, two more conditions must hold:

- After the CR-TS's recovery, the heartbeat message will be restarted immediately.
- On each recovery of the CR-TS, the starting heartbeat sequence number s is reset as 1 ($s = 1$).

Since the clocks between the FDS and the CR-TS are not synchronized, the t_r^m at the CR-TS side is not accurate from the FDS's perspective. Thus instead of piggybacking t_r^m in SET_m, SET_m contains R_m, the sending time of the m_i according to the CR-TS's local clock (σ_i) and heartbeat message m_i's sequence number i to estimate the recovery time. Thus $SET_m := \{R_m,\ \sigma_i,\ i\}$.

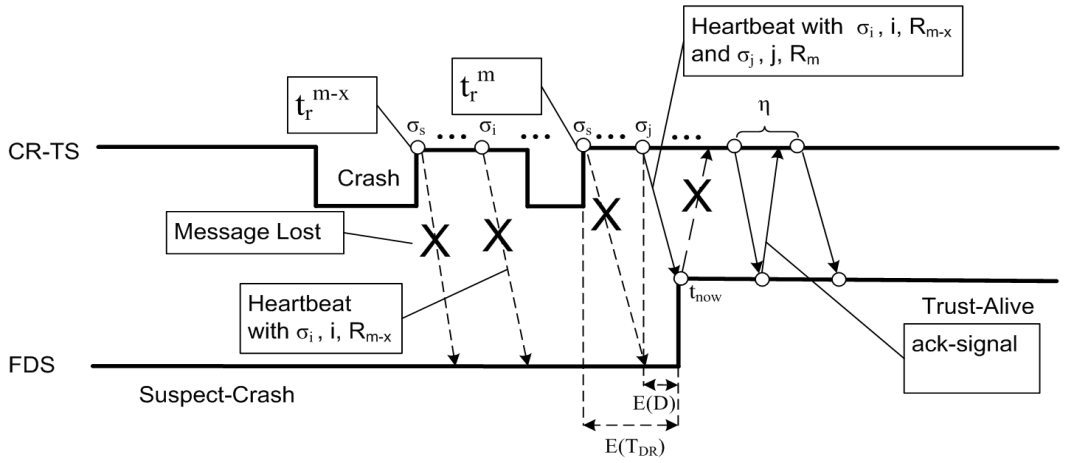


Figure 4.2: Recovery Time Estimation for the System with Unsynchronized Clocks

For the R_mth round recovery time estimation, suppose the current local time of the FDS is t_{now}, then

$$t_r^m \approx t_{now} - (i-s)\eta - E(D). \tag{4.3.14}$$

If there are more than one SET_m element within the received heartbeat message, each of the previous MTBF period's recovery time can be estimated as follows:

$$t_r^{m-x} \approx t_{now} - (\sigma_j - \sigma_i) - (i-s)\cdot\eta - E(D),\ x \geq 1. \tag{4.3.15}$$

For example, the heartbeat message m_j received by the FDS contains SET_m and SET_{m-x} due to the fact that R_{m-x} was not received by the FDS and the corresponding recovery

information (i, σ_i, R_{m-x}) is still stored in the $RECOVERY[N]$ at the CR-TS side. Although both of σ_j and σ_i are the sending time of the heartbeat messages according to the CR-TS's local clock, $\sigma_j - \sigma_i$ is an interval independent of the CR-TS's local time (see Fig. 4.2). If the drift of the CR-TS's local time is small enough to be ignored, then $t_{now} - (\sigma_j - \sigma_i) - E(D)$ can present the heartbeat message m_i's sending time according to the FDS's local clock. Therefore, the equation (4.3.15) can be used to estimate the R_{m-x}th recovery time.

Algorithm 1 A Reliable Recovery Detection Protocol for the System with Synchronized Clocks

At CR-TS Side:

1: Initialization:
2: $RECOVERY[N]$=null; $R_m = 0$;
3: for all $i \geq 1$, every η interval send heartbeat m_i to FDS; {if CR-TS crashes, the heartbeat will be stopped as well}
4: **when** CR-TS's recovery finishes, record t_r^m and R_m++;
5: $SET_m := \{R_m, t_r^m\} \rightarrow RECOVERY[N]$ {record recovery information in the persistent storage}
6: **when** heartbeat m_i is sent:
7: **if** ($Recovery[N]$!=null) **then**
8: attach all of the elements in $Recovery[N]$ to heartbeat m_i.
9: **end if**
10: **when** m_{ack} is received:
11: search and delete SET_ms in RECOVERY[N] according to the R_ms attached in m_{ack};

At FDS Side:

12: **when** heartbeat m_i is received:
13: **if** (SET_ms exist within heartbeat m_i) **then**
14: record the unrecorded t_r^ms and R_ms.
15: send acknowledgement m_{ack} attached with the received R_ms to CR-TS;
16: **else**
17: **if** (no SET_ms within heartbeat m_i $\|$ a crash failure is detected) **then**
18: stop sending m_{ack};
19: **end if**
20: **end if**

4.3.2 A Lightweight Recovery Detection Protocol

Since the recovery time estimation methods in Section 4.3.1 require persistent storage to guarantee the *strong completeness* property, the overhead generated by storage access could be high. In order to solve this problem, we propose a lightweight recovery detection protocol which does not use persistent storage is proposed. This protocol does not guarantee to detect every recovery, especially when MTTF is very short, this protocol might not detect some recoveries. But when MTTF $>> 3E(T_{DR})$, this protocol should perform reasonably well. This is because if the CR-TS's can send a reasonable number of liveness messages without crashing, the probability that the FDS receives at least one liveness message with recovery information is quite high. Actually, when the CR-TS crashes frequently, it can be regarded as an unstable period (as in [1]) and can be ignored as noise. The details of this lightweight recovery detection protocol for the system with synchronized clocks and unsynchronized clocks are given as below.

For the system with synchronized clocks, when the CR-TS recovers, the recovery time will be recorded and a heartbeat message will be sent to the FDS. In order to notify the FDS about the occurrence of the CR-TS's recovery, a recovery signal *r-signal* will be attached to the heartbeat message together with the heartbeat sending time σ_i and the heartbeat sequence number i. When the FDS receives a heartbeat message with *r-signal*, the FDS will record the recovery information within the message and send an acknowledgement message (m_{ack}) with the estimated t_r^m to the CR-TS. If the CR-TS receives this m_{ack}, the attached t_r^m will be retrieved and compared with the recorded current recovery time (t_r^c) at the CR-TS's side. If $t_r^m = t_r^c$, the message m_{ack} is for the current MTBF period. Then it will stop attaching *r-signal* to heartbeat messages. If $t_r^m < t_r^c$, then discard the message m_{ack}, because this acknowledgement message comes from a previous period. If the heartbeat message, received by the FDS, does not contain *r-signal*, then the FDS will stop sending m_{ack} and the conversation of recovery detection terminates.

In practice, even for clock synchronized systems the clocks might have small drifts and due to the different accuracy between host machines, using condition such as $t_r^m = t_r^c$

might generate mistakes. In order to tolerate such drifts or accuracy problems, a value ξ, can be defined (a positive real number, which is small enough but larger than the clocks drift and the difference of accuracy between machines). Then the condition $t_r^m = t_r^c$ can be substituted by $|t_r^m - t_r^c| < \xi$, $t_r^m < t_r^c$ can be replaced by $|t_r^c - t_r^m| > \xi$.

Since the message communication is not reliable, both of the heartbeat messages and the acknowledgement m_{ack} might be lost during transmission. If MTTF is long enough the above protocol can tolerate these message losses. This is because after the CR-TS recovers and continues sending its heartbeat without a failure, the FDS will receive a heartbeat message with *r-signal* and send m_{ack} to the CR-TS. Even if the m_{ack} might get lost as well, after some time, the CR-TS will receive a m_{ack} and stop attaching *r-signal* to the heartbeat message. Eventually, the FDS will receive a heartbeat message without *r-signal* and stop sending the m_{ack}. For the mth MTBF period, the equation (4.3.16) can be adopted for the recovery time estimation.

$$t_r^m = \sigma_i - (i-s)\eta, \tag{4.3.16}$$

where σ_i is the sending time of the message m_i; m_i is the first message received by the FDS after the CR-TS's mth recovery; s is the starting message sequence number of the mth MTBF period.

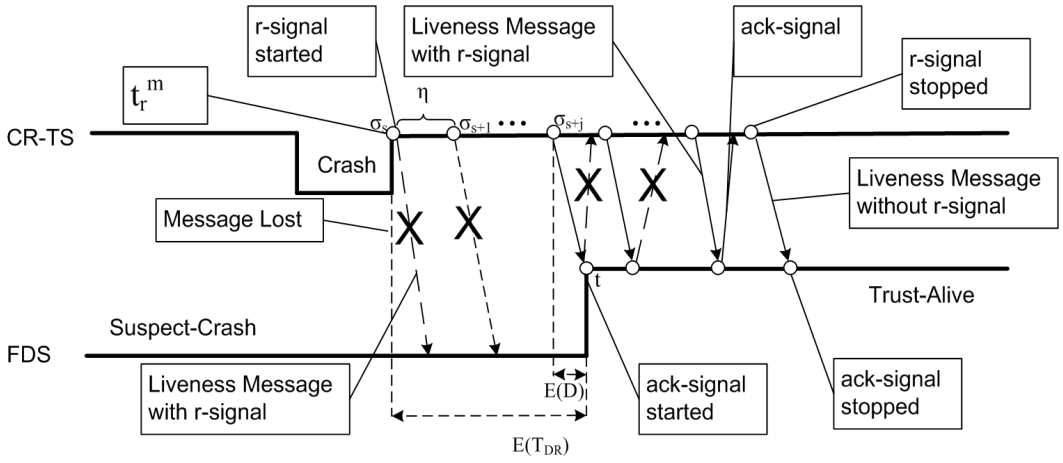


Figure 4.3: The Lightweight Recovery Detection Protocol

On average, $E(X_L)$ messages are needed for one-way message transmission. Therefore the time duration $3\big(E(D)+\eta E(X_L)\big)$ is needed for this handshake protocol.

If the local clocks between the FDS and the CR-TS are unsynchronized, then the equation (4.3.14) can be adopted for recovery time estimation. But in order to make the recovery time comparable at the CR-TS's side (e.g., $|t_r^m - t_r^c| < \xi$), the estimation of t_r^m according to the CR-TS's local clock (using the equation (4.3.16)) should be attached in the acknowledgement message as the estimated recovery time. This is because the equation (4.3.16) only needs the CR-TS's local clock for recovery time estimation. Overall, this lightweight protocol can improve the *completeness* property of the FDS, estimate the recovery time of the CR-TS and reduce the system overhead.

4.3.3 Discussion

One of the most useful aspects for such recovery detection protocols is that recovery detection protocols can be combined with failure detection algorithms to satisfy the autonomy requirement for *crash-recovery* failure detectors. Previous failure detection algorithms, such as [9, 24, 37, 51, 68, 81], cannot adapt to the *crash-recovery* behaviour of the monitored target. When a crash failure is detected, these algorithms will either terminate or still keep listening to the monitored target. If the FDS terminates after a failure is detected, then it cannot detect the recovery of the monitored target and restart the monitoring procedure again. If the FDS keeps listening, when it receives a message, it cannot separate whether the received message is generated by a recently recovered target or it is a delayed message. The recovery detection protocol enables the monitoring procedure to operate autonomously and have a consistent view of a recoverable target rather than making the *fail-free* or *crash-stop* assumption. Especially for a highly consistent *crash-recovery* service, if an existing failure detection algorithm is adopted, it cannot solve the undetectable failure problem. If all failures need to be detected, the detection time might be too small to ensure accuracy and mistake recurrence requirements. With the recovery detection protocol, these problems can be solved reasonably well. The occurrence of a failure can be implied by detecting the occurrence of a recovery. For a highly consistent *crash-recovery* service, the detection time can be scaled up within the restricted upper bound without being restricted by the required proportion of detected failures. This is because the *completeness* requirements are ensured by the recovery detection protocol rather than the failure detection

algorithm, which brings more flexibility for the failure detection algorithm adoption and more adaptivity for the highly dynamic failure detection environment.

Algorithm 2 A Lightweight Recovery Detection Protocol

At CR-TS Side:

Initialization:
  for all $i \geq 1$, every η interval send heartbeat m_i to FDS; {if CR-TS crashes, the heartbeat will be stopped as well}
when CR-TS's recovery is finished:
  int i=1; send heartbeat m_i with σ_i, i and *r-signal* to FDS; $i++$;
when an m^j_{ack} is received:
if ($|t^m_r - t^c_r| < \xi$) **then**
    stop attaching *r-signal* to m_i;
else
  if ($|t^c_r - t^m_r| > \xi$) **then** {the received m^j_{ack} is for the previous period}
      discard m^j_{ack};
  end if
end if

At FDS Side:

when m_j is received by FDS:
if (*r-signal* exists within m_j) **then**
    estimate the recovery time t^m_r;
    record recovery information;
    send m^j_{ack} attached with the estimated t^m_r to CR-TS;{t^m_r is estimated according to the CR-TS's local clock}
else
  if (no *r-signal* within m_j || a crash failure is detected) **then**
      stop sending m_{ack};
  end if
end if

4.4 Summary

In this chapter, we have introduced how to estimate the QoS of message communication and the target service's dependability metrics as input parameters for the failure detector's configuration for the system with synchronized and unsynchronized clocks in *crash-recovery* runs. Then we proposed a reliable recovery detection protocol adopting persistent storage, which guarantees to detect each recovery and a lightweight recovery detection protocol, which can reduce the system overhead. Both of these two protocols can estimate the target service's recovery time, improve the *completeness* aspect and improve the autonomy of a failure detector. Compared with the *fail-free* failure detector which assumes 100% availability and absolute reliability or the *crash-stop* failure detector which can only measure the reliability of the monitored target, a *crash-recovery* failure detector can measure all of the reliability, availability and the consistency of the monitored target more realistically.

In the next chapter, the proposed *crash-recovery* failure detection pair and the designed lightweight recovery detection protocol are simulated and evaluated under various conditions. Then the QoS of failure detectors is assessed and analyzed in depth.

Chapter 5

Simulation and Evaluation

5.1 Introduction

In this chapter, first, we present a failure detection simulation toolkit called FD-Simulator, which has been designed for studying and evaluating general failure detection frameworks and failure detection algorithms. Second, with this proposed simulator, some simulation cases are planned and performed to evaluate the QoS of the *crash-recovery* failure detector and the lightweight recovery detection protocol presented in Chapter 3 and 4, respectively. Then, we plot both the analytical results derived from Theorem 3.6.2 and the simulation results according to the proposed simulation cases. Both the analytical results and the simulation results show that the dependability of the monitored target has impact on the QoS of failure detectors, particularly when the monitored target is highly consistent but not highly reliable. In addition, the proposed lightweight recovery detection protocol can improve the *completeness* of a failure detector without reducing other QoS aspects, which is useful for the highly consistent but unreliable service monitoring and can enable the FDS to operate autonomously. Finally, the chapter finishes with some conclusions.

5.2 Failure Detection Simulator

5.2.1 Motivation for Implementing a Simulator

Originally, we tried to implement a service-oriented failure detection framework based on the Globus Open Grid Service Architecture (OGSA) [55] and the OGSI 1.0 specification [83] as a subcomponent of the DIGS project. We adopted Globus Toolkit 3 [73] as our implementation platform. During the implementation, several problems occurred:

- As emerging technologies, Web Service and Grid standards change dramatically. Particularly for Grid standards, the OGSA framework and OGSI specification expired in 2004 and to be substituted by WSRF and the consequent specifications. This makes the implementation work more difficult.
- As our final aim was to provide a failure detection framework solution for large-scale distributed systems, deploying multiple failure detectors could be time-consuming work. However, we could not yet know the performance of such a failure detection framework and the corresponding algorithms before we implement all of them. Thus implementing a failure detection framework without empirical simulation is a potential risk from the design and implementation perspective.
- The performance evaluation of an implemented failure detection framework and the corresponding algorithms could be difficult, since performance data, such as the QoS of failure detection services or resource overheads, are not easily collected within a large-scale distributed system.
- Injecting failures into monitored applications could be a difficult and time-consuming task. Producing various types of failures and injecting them into implementation code is complicated.
- Performance evaluation based on a test implementation might only evaluate the QoS of a failure detection framework under a particular environment rather than evaluate a more general solution, which cannot guarantee that a designed frame-

work or algorithm is a good solution for any circumstance.

Given the above problems, we decided to translate the implementation code into simulation code in order to study the failure detection problem in a flexible experimental environment.

5.2.2 Related Work

In this section, we review the variety of simulators which are currently available and highlight why none of them was suitable for our purpose. We group the simulators according to their focus and introduce each of them briefly in terms of their design aims as follows.

Network Simulators

There are a number of simulators, which can simulate large-scale networks, such as

- ns2 Network Simulator [66] is a C++/TCL based simulation tool, which provides a discrete event-based simulation environment. The main aim of ns2 is to simulate network topology or transport level protocols. Although ns2 can construct a large-scale network easily, it only focuses on simulating network level problems rather than higher level application's.
- GloMosim [6] is a simulation environment for scalable networks and provides an application level API, but it is especially designed for wireless networks.

Grid Simulators

There are serval simulators for Grid systems simulation, but most of these tools focus on resource usage and are used to study scheduling problems.

- GridSim [16] is a SimJava-based [52] simulation tool to evaluate scheduling algorithms, which allows the user to model and simulate parallel or distributed systems. GridSim provides a comprehensive API for creating different types of resources such as parallel machines or clusters etc. GridSim is a convenient tool to simulate how a resource broker aggregates Grid resources according to

their availability to meet the user's or applications requirements using scheduling algorithms or policies.

- SimGrid [19] is a scalable, extensible discrete event simulation toolkit for evaluating the distributed and parallel application scheduling on distributed computing platforms, which is implemented in C. SimGrid enables the user to model various types of resources such as CPU, machine etc. and allows one to model more realistic Grid topologies and bandwidth sharing. SimGrid also provides analysis of the performance of scheduling algorithms and the resource usage.
- Optorsim [8] is a Java-based simulator to evaluate a data Grid. Optorsim mainly focuses on analyzing data management policies, various replication optimization algorithms and evaluating the impact of the choice of replication algorithms on the throughput of Grid jobs.
- VOGanglia [39] aims to provide a simulation platform for cluster scheduling problems. The VOGanlia simulator provides support for the analysis of different scheduling policies in a multiple virtual organization environment.
- MicroGrid [79] aims to provide a simulation environment to systematically study and evaluate middleware applications, network services for the computational Grid. MicroGrid is implemented in C++ and focuses on simulating Grid resource management problems within a virtual organization, which requires a cluster-level system environment and related software.

Dependability Simulators

In addition, there are also some dependability modeling and simulation assessment tools such as:

- Möbius [30], is a system-level dependability and performance modeling toolkit implemented in C to assess the system dependability. It provides a discrete event simulator module to enable models to be simulated in a distributed way across multiple machines to speed-up the simulation.
- Rainbow Net Simulator [74] is a Petri Net-based simulation tool, which combines graphical and state variable-based specification for discrete event simula-

tion. Rainbow enables the user to capture a realistic system behavior model.

- DEPEND [42] is a functional-based system-level simulation tool to simulate realistic fault models for system dependability analysis.

Distributed Algorithm Simulator

- Neko [84] is a Java-based simulation platform, which is based on SimJava originally (now based on NekoSim). The main aims of Neko are to evaluate and analyze the performance of distributed algorithms such as consensus, group membership, failure detection. A simulation constructed using Neko can be either performed on a simulated network environment or a real network environment. However, Neko focuses on the distributed algorithm level without considering underlying resources and Neko is more suitable for flat distributed algorithm simulation rather than hierarchical protocols or frameworks.

Although there are a number of simulators, as we described above, none of them fits our aims and purpose. Thus, we implemented a simulation tool called FD-Simulator which adopts SimJava as the simulation engine. The reason that we adopted SimJava is that SimJava is flexible and provides the basic functions we need. It is also adopted by both resource level Grid simulators such as GridSim and algorithm level simulators such as Neko. This provides the possibility to connect our FD-Simulator to GridSim and Neko to produce a more complicated resource-algorithm-architecture-dependability simulation in the future. SimJava provides a simulation kennel for entity-based discrete event simulation, an animation module to visualize the simulation as a graphical applet, a statistical package which provides statistical functions and a graphical tool for visualizing the statistical report. However, SimJava does not provide resource or application level abstractions to simplify the simulation. Further coding work is necessary to develop a task specific simulation model. Thus FD-Simulator is designed as a toolkit particularly for evaluating the distributed failure detection frameworks and the failure detection algorithms in various environments.

5.2.3 FD-Simulator Design Features

As we explained above, the existing simulation tools (see Section 5.2.2) did not provide all the expected functions, such as application architecture simulation, algorithm simulation, resource simulation, dependability assessment etc., to study failure detection problems, a simulation tool called FD-Simulator was designed and implemented specifically for studying the failure detection problems. Fig. 5.1 shows the architecture

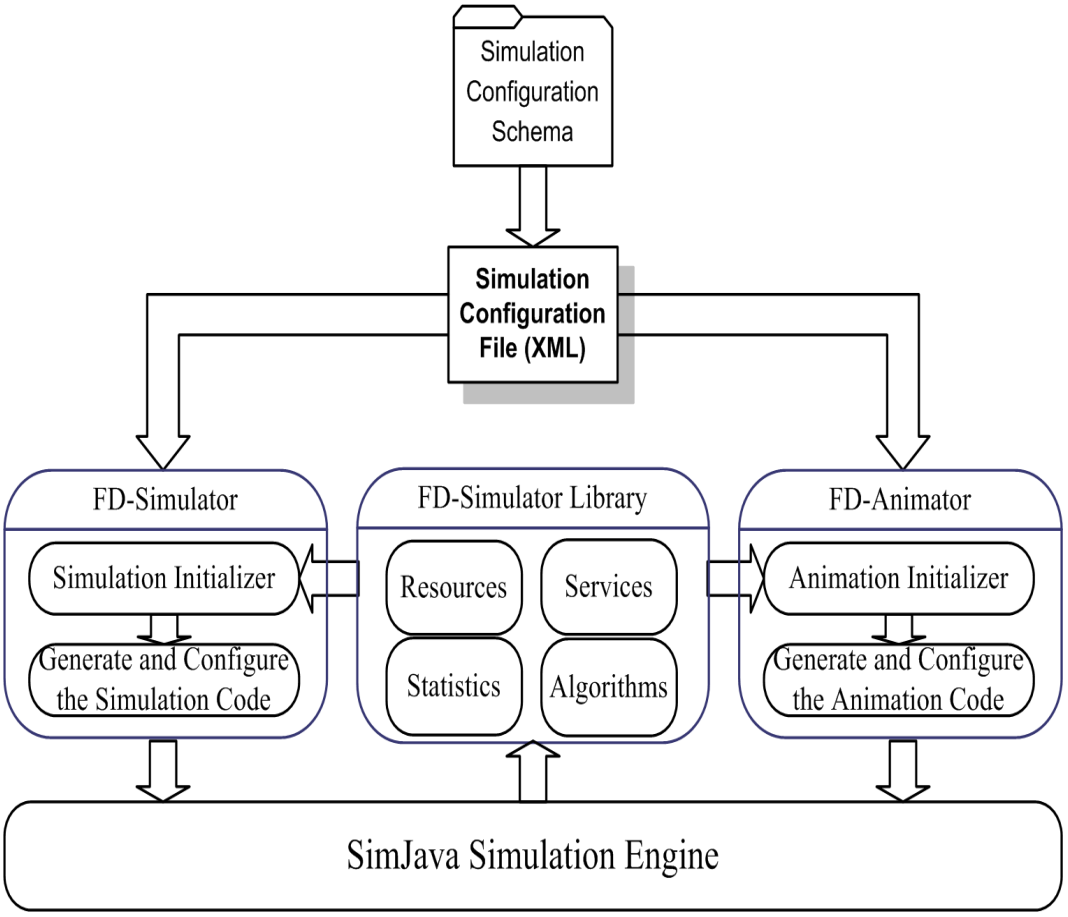


Figure 5.1: The FD-Simulator Architecture

of the FD-Simulator, which aims to provide the following features:

- Evaluating the QoS of failure detection algorithms, protocols or frameworks.
- Assessing the reliability, availability, scalability, adaptivity etc. of the proposed failure detection algorithms, protocols or frameworks.

- Simulating hierarchical failure detectors as well as flat failure detectors.
- Simulating and tracking the resource usage such as network bandwidth and CPU workload, to study the trade-off between the QoS and the system overhead.
- Minimizing the coding work and providing statistical functions for the QoS analysis of the proposed failure detectors.

Fig. 5.1 shows the architecture of the FD-Simulator, which contains predefined classes such as *Network*, *Host*, *Service* etc., some predefined algorithms and statistical tools to support the simulation. The FD-simulator is designed and implemented with the aim of minimizing the coding work to conduct simulations. Each simulation can be described by a programming language independent XML configuration file validated by a W3C conformed schema [86] file (see Fig. 5.2). When an algorithm is tested with different parameters or within a different framework, we only need to revise the configuration file and restart the simulation. This avoids rewriting and recompiling the source code. It is also simplifies the simulation procedure and makes the FD-Simulator much easier to use. When the simulation starts, the simulation initializer will read the specified configuration file to construct the network and the failure detection framework topology automatically and configure each component according to specified parameters within the configuration file.

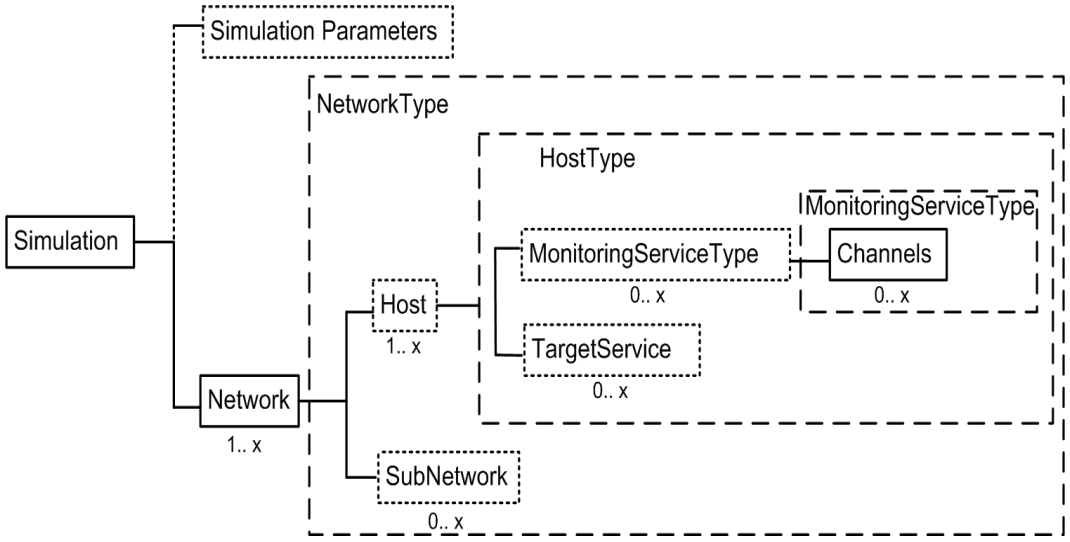


Figure 5.2: The Schema of the Configuration File

```
<xs:complexType name="EntityType">
    <xs:attribute name="failurerate" />
    <xs:attribute name="failure_variance" use="optional" />
    <xs:attribute name="mttr_distribution" type="xs:string" use="optional" />
    <xs:attribute name="mttf_distribution" type="xs:string" use="optional" />
    <xs:attribute name="type" type="xs:string" />
    <xs:attribute name="id" type="xs:string" />
    <xs:attribute name="name" type="xs:string" />
    <xs:attribute name="recoveryrate" type="xs:double" />
    <xs:attribute name="recovery_variance" use="optional" />
    <xs:attribute name="statetrace" type="xs : boolean" use="optional" />
</xs : complexType>
```

Figure 5.3: The Schema of the EntityType

Fig. 5.2 shows the schema file's structure, which is used to validate and constrain the configuration files' format. The root node of the configuration file is the *Simulation* node, which contains simulation parameters and *Network* nodes. The *Network* node contains *Subnetworks* and *Hosts*. The structure of the *Subnetwork* node is the same as the *Network* node and the *Host* node can contain *MonitoringService*[1] and the monitored *TargetService*. Each *MonitoringService* contains a group of *ChannelModules*. Each *ChannelModule* contains one *TargetService*'s name and some corresponding monitoring parameters (see Fig. 5.6). During the initialization procedure each *MonitoringService* will create a number of channels specified in its configuration file and initialize each channel with the specified parameters. If there is no channel information within the *MonitoringService*, the *MonitoringService* will not monitor any target.

[1]Monitoring is used instead of failure detection to adapt to the potential future function extension.

Within the configuration file, the *EntityType* (see Fig. 5.3) is the fundamental schema element type. All of the other types such as *Network*, *Host*, *Service* inherit from the *EntityType* element. The *EntityType* has some predefined attributes. For example, the *mttf_distrbution* attribute represents the probability distribution of MTTF; *mttr_distrbution* attribute represents the probability distribution of MTTR; the *failurerate* attribute represents $\frac{1}{\text{MTTF}}$; the *recoveryrate* attribute represents $\frac{1}{\text{MTTR}}$. If the *failurerate* of an entity is zero, it means the entity is set as *fail-free*. If the *recoveryrate* of the entity is zero, it means the entity is set as *crash-stop*. If a probability distribution is not specified, then the FD-Simulator will regard the "rate" (e.g., *failurerate*) as a deterministic number rather than a probabilistic one.

According to the structure of the configuration schema file, some important classes are implemented as the default library of the FD-Simulator. In this thesis, we give a brief introduction of the most important classes. The foundation class within this simulator is the *SimEntity* class, which corresponds to the *EntityType* element within the schema file. Other entity classes such as *Network*, *Host*, *TargetService*, *MonitoringService*, *ChannelModule* etc. all inherit from the *SimEntity* class. The *SimEntity* contains the basic functions used for the simulation. For example, the functions for simulating a crash, recovery, sending a message, receiving a message etc. The subclasses that inherit from the *SimEntity* class can use or extend these functions for their own purposes. In our simulation, a crash is simulated as the crashed entity keeping silent and not sending or replying to any message; a recovery is simulated as the recovered entity coming back to life from a crashed state with the ability to send and reply to any message. The crash or recovery can be scheduled as discrete events within the entity with a given arrival rate and a specified random distribution. When the crash event arrives, the crash function will be triggered and a recovery event will be scheduled; when the recovery event arrives, the recovery function will be triggered and a crash event will be scheduled. In this way, a *crash-recovery* model can be simulated. When the crash arrival rate equals zero (MTTF $\rightarrow \infty$), a *fail-free* run can be simulated; when the recovery arrival rate equals zero, a *crash-stop* run can be simulated.

Another key component for the failure detection simulation is the simulated network entity. From the network perspective, each network entity can be regarded as a queue.

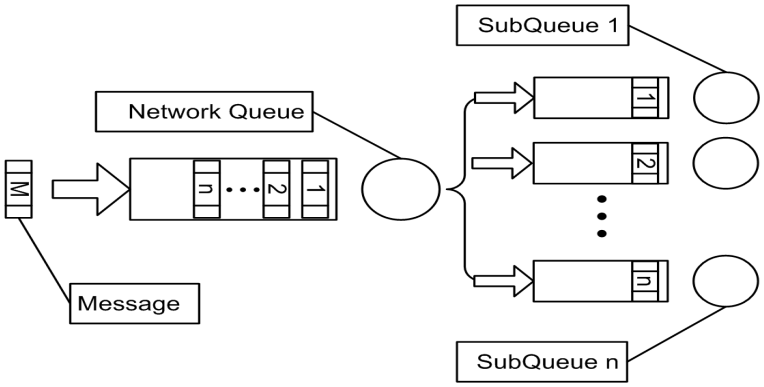


Figure 5.4: The Simulation Network Queue Model

But a potential problem of using a single queue to simulate a network entity is that when the message arrival rate (from multiple sources) is high, the queue might become unstable and the arriving messages could possibly be congested in the queue. In order to control such message congestion, we adopted multiple queues (see Fig. 5.4) to simulate a network entity. In the FD-Simulator, a network entity contains two types of queue. One type is the service queue (the *SubQueue* in Fig. 5.4), which will hold the arriving message for a given time (deterministic or non-deterministic), then delivers it to the destination. The other type is the control queue (the *NetworkQueue* in Fig. 5.4), which will deliver the arriving message to a service queue and the service time of the *NetworkQueue* is sufficiently small to be ignored. A *Network* can contain one control queue and several service queues. The number of service queues can be specified, so that the level of message congestion is controllable. For example, if each message's transmission is assumed to be independent, then a network should have a reasonable number of service queues to satisfy the independence assumption. When a message arrives at a network, it will be received by the control queue first, then the control queue will deliver the message to the service queue next to the previous one, which services the last arrived message, thus allowing parallel transmission of each message. Then, after some service time, the message will be delivered to its next destination. By adopting such a method, the message congestion and dependency can also be controlled by the simulator to satisfy a given message transmission assumption. For other entities, such as the *Host* entity, if necessary, they can also be designed in such a way to satisfy the simulation requirements.

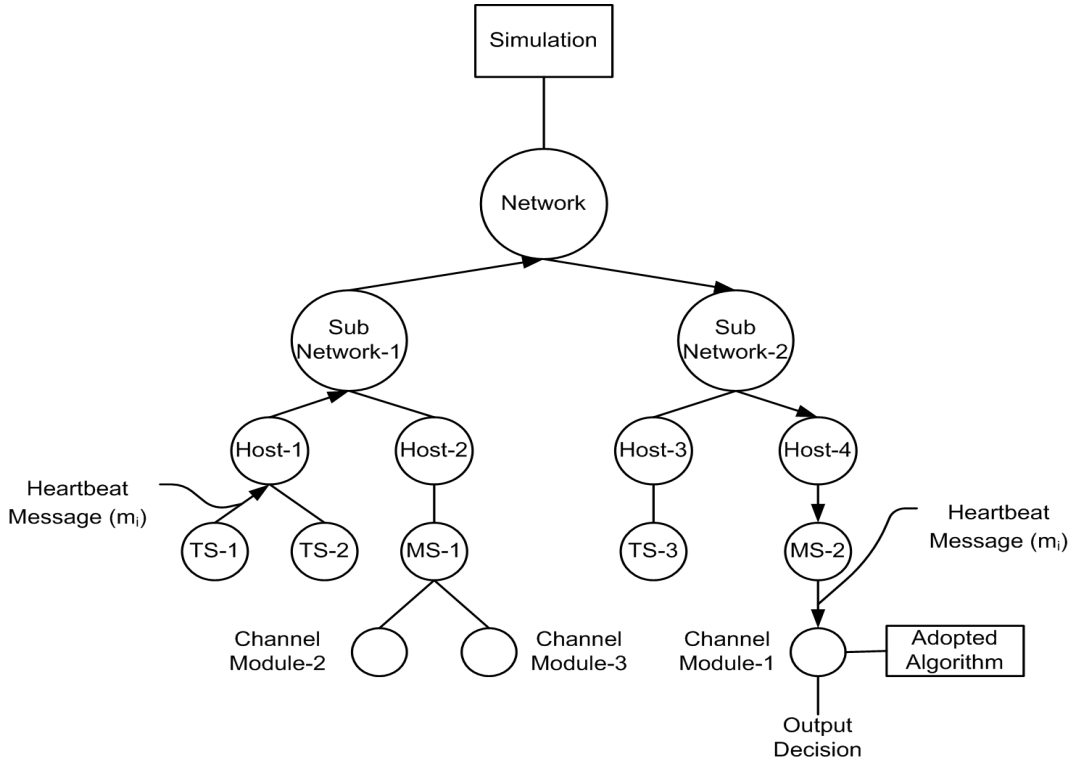


Figure 5.5: A Sample Failure Detection Simulation Framework

Fig. 5.5 shows a sample simulation topology according to a configuration file and how the simulation works. The root node of the XML-based simulation configuration file is *Simulation*, which can contain one *Network* node and some simulation parameters. A *Network* node can contain some other *Network* nodes as its subnetwork or *Host* nodes. A *Host* node can contain some *TargetService* (TS) nodes or *MonitoringService* nodes (MS). For the *MonitoringService* node, it contains *ChannelModule*s. Each *ChannelModule* can monitor a *TargetService* and detect its failure by adopting some algorithm. When the simulation is started, the FD-Simulator will load the configuration information from the specified configuration file, generate the system topology according to the structure of the configuration file, and configure each node according to the given parameters in the configuration file. When a *MonitoringService* is specified to monitor a *TargetService*, a *ChannelModule* should be paired with the *TargetService*'s name. When the simulation starts, the *TargetService* will be configured automatically to communicate with the paired *ChannelModule*

```
<xs:attribute name="interval" type="xs:double" />
<xs:attribute name="timeout" type="xs:double" />
<xs:attribute name="algorithm" type="xs:string" />
<xs:attribute name="protocol" type="xs:string" />
<xs:attribute name="source"use="xs:string" />
<xs:attribute name="dest" type="xs:string" />
```

Figure 5.6: Parameters for a ChannelModule

and the *ChannelModule* will monitor the *TargetService* by adopting a given algorithm that can give an output decision of the *TargetService*'s status. For example, if a *push-style* crash failure detector is simulated as in Fig. 5.5, the TS-1 is monitored by the ChannelModule-1. When the simulation starts, TS-1 starts to send heartbeat messages with the final destination as ChannelModule-1. The heartbeat message will route from TS-1 to Host-1, SubNetwork-1, Network, SubNetwork-2, Host-4, MS-2, then finally reaches ChannelModule-1. This routing method is implemented by adopting XPath [27] query. When a message is routing, the current message hosting node will query itself and all of its sub nodes whether the final destination of the message is within this node or its sub nodes. If yes, it will deliver the message to the node which contains the message's destination; if not, the message will be delivered to the parent node of the current node then the query procedure restarts. In such a way, a message can be delivered to any node in the simulated network system, complicated hierarchical failure detection frameworks can be simulated and various algorithms can be designed and injected into each *ChannleModule* for QoS evaluation.

The FD-Simulator also provides an animation feature (see Fig. 5.1). When the animation feature is enabled in the configuration file, the FD-Animator can initialize an applet-based animation and construct the graphical network topology automatically

rather than starting a non-graphical simulation. This animator can help with debugging the simulation code, verifying the behaviors of a framework, an algorithm or simulation results during development time. It can also demonstrate the monitoring procedure and help understanding of the proposed failure detectors.

Moreover, the simulation parameters, such as the simulation runtime, which represent the length of the simulation; the warmup duration, which represents the duration before the simulation reaches steady state; simulation replication numbers, confidence interval, produce trace file etc., can be specified before the simulation starts in the XML-based configuration file (see Fig.5.7). Additional parameters can easily be defined and added, which brings more flexibility to the simulation configuration.

5.2.4 Simulator Summary

In the previous sections we introduced the design and implementation of our FD-Simulator and gave a survey of the related work. The FD-Simulator can conduct and configure a failure detection framework automatically according to a given XML-based configuration file. The XML-based configuration presents the hierarchical architecture of a simulated failure detection network effectively and simplifies the coding work of evaluating the given failure detection framework and the failure detection algorithms. The QoS data can be gathered after the simulation finishes and statistical functions are provided to allow detailed analysis. Furthermore, the FD-Simulator also supports an animation feature and can visualize the failure detection framework, which can help in validating, verifying and understanding the implemented algorithm and the communication procedure between the failure detection pairs.

Using the FD-Simulator, some simulation cases have been planned and simulated to evaluate the QoS of failure detectors with some algorithms under various conditions. These are presented in the following sections.

```
<xs:element name="SimulationParameters">
  <xs:complexType>
  <xs:all minOccurs="0">
    <xs:element name="runtime" type="xs:double" />
    <xs:element name="warmuptime" type="xs:double" />
    <xs:element name="statistics" type="xs:boolean"minOccurs="0" />
    <xs:element name="graphstatistics" type="xs:boolean"minOccurs="0" />
    <xs:element name="animation" type="xs:boolean"minOccurs="0" />
    <xs:element name="showmessage" type="xs:boolean"minOccurs="0" />
    <xs:element name="trackfile"minOccurs="0" />
    <xs:element name="report" type="xs:boolean"minOccurs="0" />
    <xs:element name="statetrace" type="xs:boolean"minOccurs="0" />
    <xs:element name="reportdetail" type="xs:boolean"minOccurs="0" />
    <xs:element name="includetransient" type="xs:boolean"minOccurs="0" />
    <xs:element name="autotrace" type="xs:boolean"minOccurs="0" />
    <xs:element name="generategraph" type="xs:boolean"minOccurs="0" />
    <xs:element name="print_to_std" type="xs:boolean"minOccurs="0" />
    <xs:element name="replication_number"type="xs:int"minOccurs="0" />
    <xs:element name="confidence_interval"type="xs:double"minOccurs="0" />
  <xs:all>
<xs:complexType>
```

Figure 5.7: Simulation Parameters

5.3 Simulation Planning

In the previous section the FD-Simulator was introduced for evaluating failure detection frameworks and failure detection algorithms. In the following sections, the simulation results will be introduced, which were produced using the FD-Simulator. The simulations are made according to the system model described in Chapter 3, which contains a failure detection pair (CR-TS and FDS), hosted by two hosting machines respectively within a network (see Fig. 5.8).

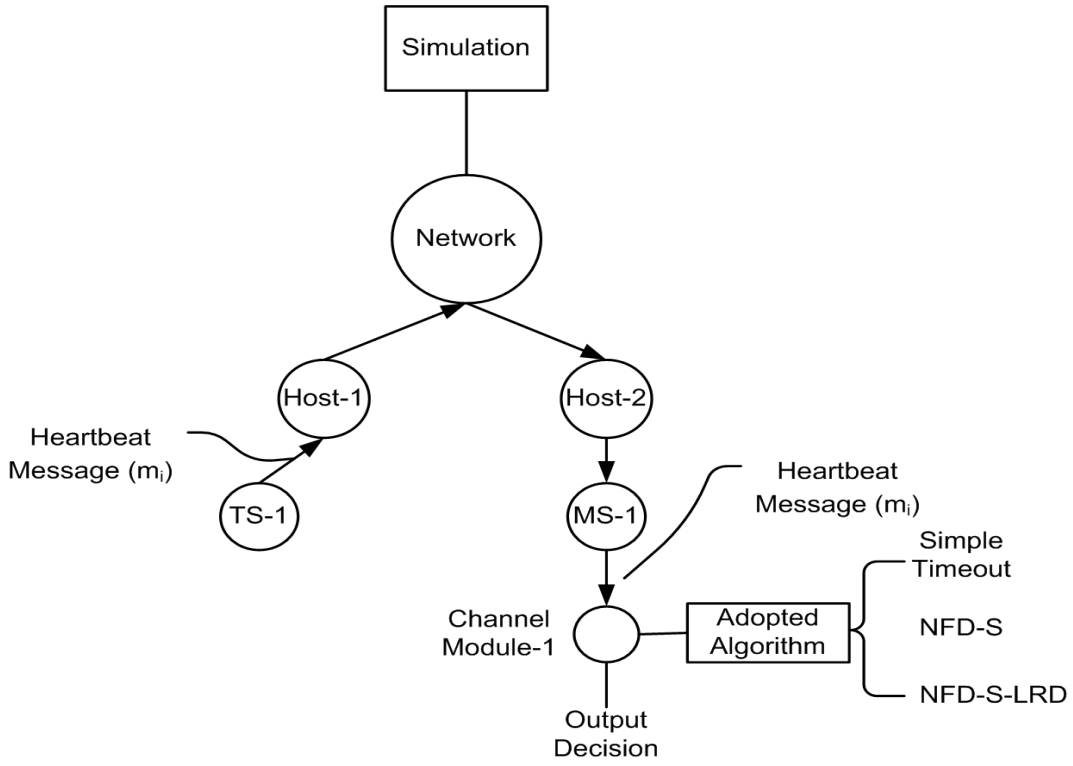


Figure 5.8: The Planned Failure Detection Simulation Framework

The designed simulation plan has eight simulation cases (see Table 5.1). The planned simulation cases are distinct from previous simulation studies such as in [24, 36, 68, 81], all of which focus on evaluating how well a failure detector can tolerate message delay and loss behavior under the *fail-free* or *crash-stop* assumption. Our simulation cases are focusing on the impact of the monitored target's *crash-recovery* on the QoS of a failure detector and how well a failure detector adapts to such unavoidable events.

The simple heartbeat timeout algorithm, the NFD-S algorithm and the NFD-S algorithm with the lightweight recovery detection protocol (NFD-S-LRD) are tested with various values of MTTF and MTTR. Table 5.1 shows the simulation parameter settings of the simulations.

FDS		CR-TS			Network	
Algorithm		Reliability	Consistency	Availability	p_L	$E(D)$
Simple Timeout	1	MTTF = 100 (Exponential)	MTTR = 5 (Exponential)	$\frac{\text{MTTF}}{\text{MTBF}} \approx 0.9524$	0.01	0.02
	2	MTTF = 1000 (Exponential)	MTTR = 50 (Exponential)	$\frac{\text{MTTF}}{\text{MTBF}} \approx 0.9524$	0.01	0.02
	3	MTTF = 100 (Deterministic)	MTTR = 5 (Deterministic)	$\frac{\text{MTTF}}{\text{MTBF}} \approx 0.9524$	0.01	0.02
	4	MTTF = 1000 (Deterministic)	MTTR = 50 (Deterministic)	$\frac{\text{MTTF}}{\text{MTBF}} \approx 0.9524$	0.01	0.02
NFD-S	5	MTTF = 100 (Exponential)	MTTR = 5 (Exponential)	$\frac{\text{MTTF}}{\text{MTBF}} \approx 0.9524$	0.01	0.02
	6	MTTF = 1000 (Exponential)	MTTR = 50 (Exponential)	$\frac{\text{MTTF}}{\text{MTBF}} \approx 0.9524$	0.01	0.02
NFD-S-LRD	7	MTTF = 100 (Exponential)	MTTR = 5 (Exponential)	$\frac{\text{MTTF}}{\text{MTBF}} \approx 0.9524$	0.01	0.02
	8	MTTF = 1000 (Exponential)	MTTR = 50 (Exponential)	$\frac{\text{MTTF}}{\text{MTBF}} \approx 0.9524$	0.01	0.02

Table 5.1: The Simulation Plan

For simulation parameter settings, we fix the heartbeat interval $\eta = 1$ and increase the *timeout* length gradually. The message transmission parameters are p_L=0.01, $E(D) =$ 0.02 as two exponentially distributed random variables respectively. The confidence interval is set as 99% [2]. All of these settings are similar to those in the simulations in [24]. Furthermore, the CR-TS (*TargetService*) is defined as a recoverable process with various MTTF and MTTR as deterministic and non-deterministic values. For

[2]The graphs are shown without confidence intervals to avoid obscuring the graphs.

the non-deterministic MTTF and MTTR, the exponential distribution is adopted. We choose the exponential distribution for the non-deterministic MTTF and MTTR for the following reasons: first, exponential failures are widely adopted for reliability analysis in many practical systems; second, unlike some heavy tail distributions such as log-normal distribution, the crash and the recovery with exponential distribution will occur with a reasonable inter-arrival time, which will not make the CR-TS behave like a *fail-free* or *crash-stop* service. In addition, some reasonable durations of MTTF and MTTR are provided as simulation cases, which are shown in Table 5.1. From Lemma 3.4.2, we know that the availability of the CR-TS is $\frac{\text{MTTF}}{\text{MTBF}} \approx 0.9524$ and the reliability is 100 or 1000 time units. Such characteristics of the CR-TS are typical for a highly available and consistent, but not highly reliable, service. The simulations with MTTF $= 100$ and MTTR $= 5$ are simulated with $100,000$ time units, with the first $2,000$ time units removed as the transient period. In order to generate enough crashes and recoveries, the simulations with MTTF $= 1000$ and MTTR $= 50$ are simulated with $300,000$ time units, with the first $60,000$ time units as the transient period. Furthermore, in order to improve the accuracy of the simulation, for each simulation case, three independence simulation replications are performed simultaneously and the average simulation results of all simulation replications are taken as the final results. The *timeout* length is varied from 1 to 20 for the simple timeout algorithm and from 0 to 20 for both the NFD-S and NFD-S-LRD algorithms, respectively. Finally, we gather the following QoS measurements: the average mistake duration, the probability of accuracy[3] and the mistake recurrence for the simple algorithm. For the NFD-S and NFD-S-LRD algorithms, we also obtain the average failure detection time and the proportion of the detected failures. In addition, we plot the average recovery detection time for the NFD-S-LRD. For the other QoS metrics, since they can be derived directly from the above metrics, we do not plot them in this chapter.

[3]We use the total proportion of the time that the FDS is in *Accurate* state (see Fig.3.4(b) and Table 3.1) instead of $P_A = 1 - \frac{E(T_M)}{E(T_{MR})}$ to obtain P_A. Our tests show that the result of both kinds of calculations are very similar, when the runtime is long enough.

5.4 Simulation Evaluation and Analysis

5.4.1 Evaluation of the Simple Timeout Algorithm

In order to assess the impact of the CR-TS's dependability on the QoS of the FDS, first of all we have conducted simulation case 1 in the Table 5.1: a heartbeat algorithm which, whenever a heartbeat is received or a *timeout* is reached, resets the next *timeout* period. We choose exponentially distributed MTTF and MTTR of 100 and 5 respectively, a typical CR-TS, which is highly available and consistent but not highly reliable. The simulation results in MTTF $= 105.962$ and MTTR $= 4.937$. Then From Proposition 3.4.1 and Lemma 3.4.2 we calculate that the MTBF $= 110.899$ and the availability of this CR-TS: $P_a = \frac{\text{MTTF}}{\text{MTBF}} = 0.95548$.[4]

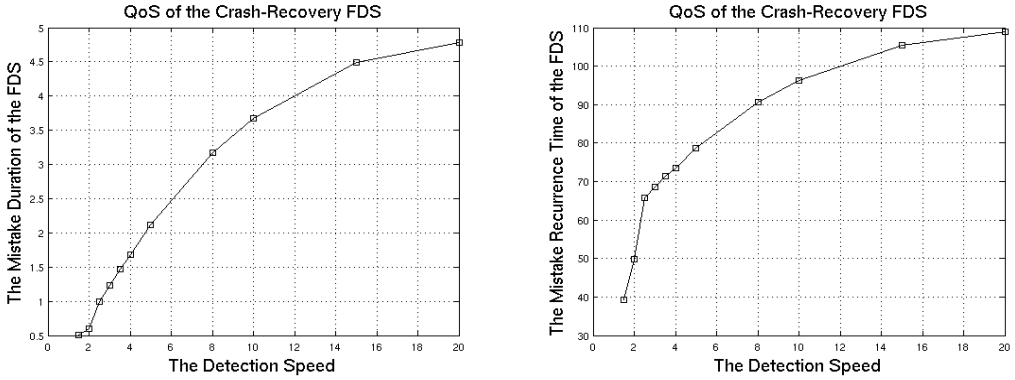
Fig. 5.9 shows the $E(T_M)$, $E(T_{MR})$, P_A changes when the *timeout* period is increased to four times MTTR. These simulation results corroborate our discussion in Section 3.9. $E(T_M)$ approaches MTTR $= 4.7803$; $E(T_{MR})$ approaches MTBF $= 108.9127$; P_A increases a little but then decreases to $\frac{\text{MTTF}}{\text{MTBF}} = 0.9563$. Note that if MTTR were increased, P_A will increase for longer and decrease more slowly (see Fig.5.13).

Then according to the simulation plan in Table 5.1, we also simulated and plotted the simulation cases 2-4. We varied the *timeout* from 1 to 20, and then observed the QoS metrics of the FDS. Figs. 5.10, Fig. 5.12 and Fig. 5.13 show the performance of $E(T_M)$, $E(T_{MR})$ and P_A of the simulation cases 1-4.

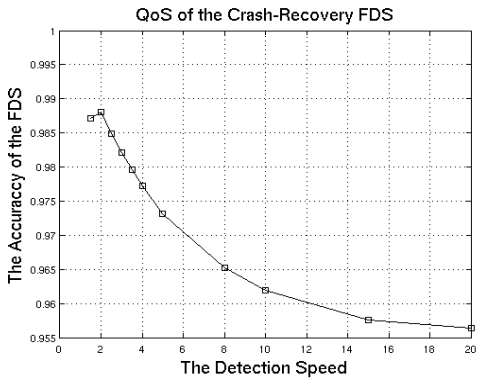
Fig. 5.10 demonstrates the $E(T_M)$ of the simple timeout algorithm. The simulation cases 1 and 3 (MTTR $= 5$) show that for both deterministic and exponential MTTR, as the *timeout* length increases, eventually $E(T_M)$ will be become MTTR, which means that $E(T_M)$ is bounded by MTTR. But the deterministic and non-deterministic simulation cases exhibit different characteristics. When *timeout* $<$ MTTR, for the simple timeout algorithm, $E(T_M)$ of the deterministic and exponential MTTR are similar.

When *timeout* $>$ MTTR, $E(T_M)$ of the deterministic MTTR increases faster than in

[4] We set MTTF=100, MTTR=5 for the simulation. Since the crash and recovery durations are generated as exponentially distributed random numbers, the discrepancy exists between the set values and the result values.



(a) $E(T_M)$ (b) $E(T_{MR})$



(c) P_A

Figure 5.9: QoS of the Crash-Recovery FDS

the exponential one. This is because, after *timeout* exceeds MTTR ($T_D^U >$ MTTR), all crashes will become undetectable in the deterministic case. Thus $E(T_M)$ will approach MTTR rapidly. But for exponential MTTR, the proportion of the detectable crashes will decrease more gradually. Thus $E(T_M)$ approaches MTTR more slowly than in the deterministic case.

Simulation cases 2 and 4 (MTTR $= 50$) show that if MTTR becomes large, as the *timeout* length increases, $E(T_M)$ can also become larger as well. This is because the upper bound of $E(T_M)$ increases. Note that compared with the simulation cases 1 and 3, the simulation cases 2 and 4 in Fig. 5.10 only show the linear part rather than the complete characteristics. If the *timeout* length increases to 200, the simulation cases 2 and 4 will exhibit the same shape as the simulation cases 1 and 3. We can observe

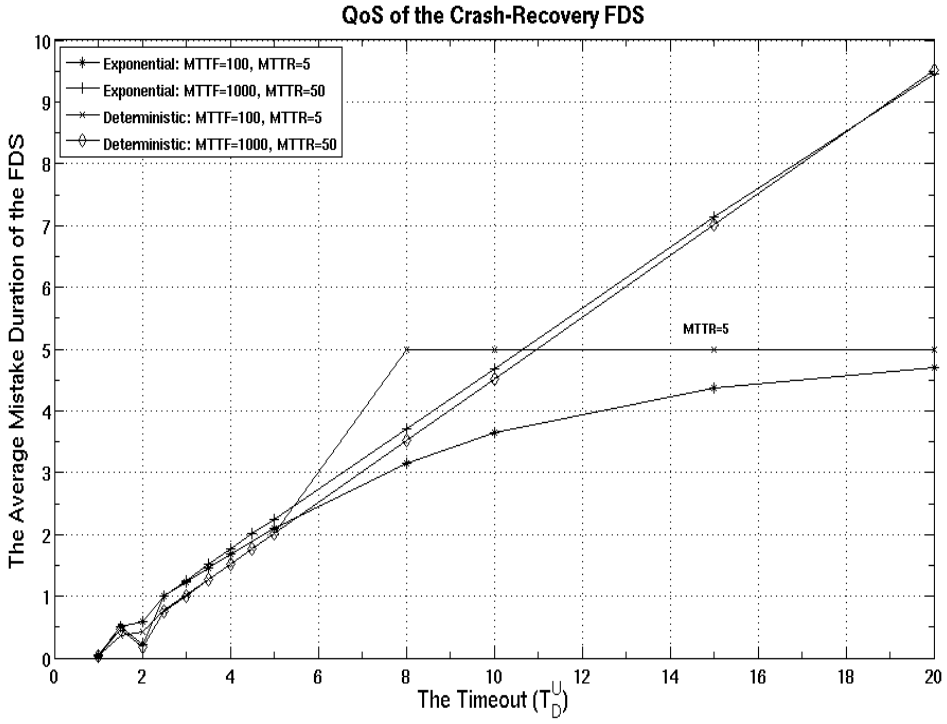


Figure 5.10: The Simple Timeout Algorithm: $E(T_M)$

that, for all of the simulation cases 1-4, $E(T_M)$ decreases (or increases slower) from $1.5 - 2.1$ and then increases again. We analyze this phenomenon in detail as follows.

As we discussed in Section 3.6.2, in a *crash-recovery* run, mistakes caused by the CR-TS's crash and recovery will be taken into consideration. Therefore, for the same *timeout* length, there are four aspects which have impact on T_M: the message delay and loss, the CR-TS's crash and recovery (see Fig. 5.11). T_M caused by a message delay is governed by the ratio between $E(D)$ and T_D. For the same $E(D)$, as *timeout* increases, more and more delayed messages can be tolerated. Thus T_M caused by a message delay (T_M^1) will decrease and occur less frequently. T_M caused by a message loss (T_M^2) is related to η, p_L, $E(D)$ and the *timeout* length. For constant message communication QoS (i.e. the same p_L and $E(D)$), T_M caused by message loss is governed by the ratio between η and T_D. Since as the *timeout* length increases, more and more message losses can be tolerated, the average duration of T_M^2 will decrease and T_M^2 will occur

less frequently (see Fig. 3.10). T_M caused by a crash (T_M^3) is mainly governed by T_D (see Fig. 5.11(c)), because if a crash occurs, a *false positive* mistake will last until the *timeout* time or until the CR-TS recovers. For detectable crashes, as the *timeout* length increases, T_M^3 will increase. T_M caused by a recovery (T_M^4) is mainly governed by p_L and $E(D)$ (see Fig. 5.11(d)), since after the CR-TS's recovery, a recovery can be detected when a liveness message is received.

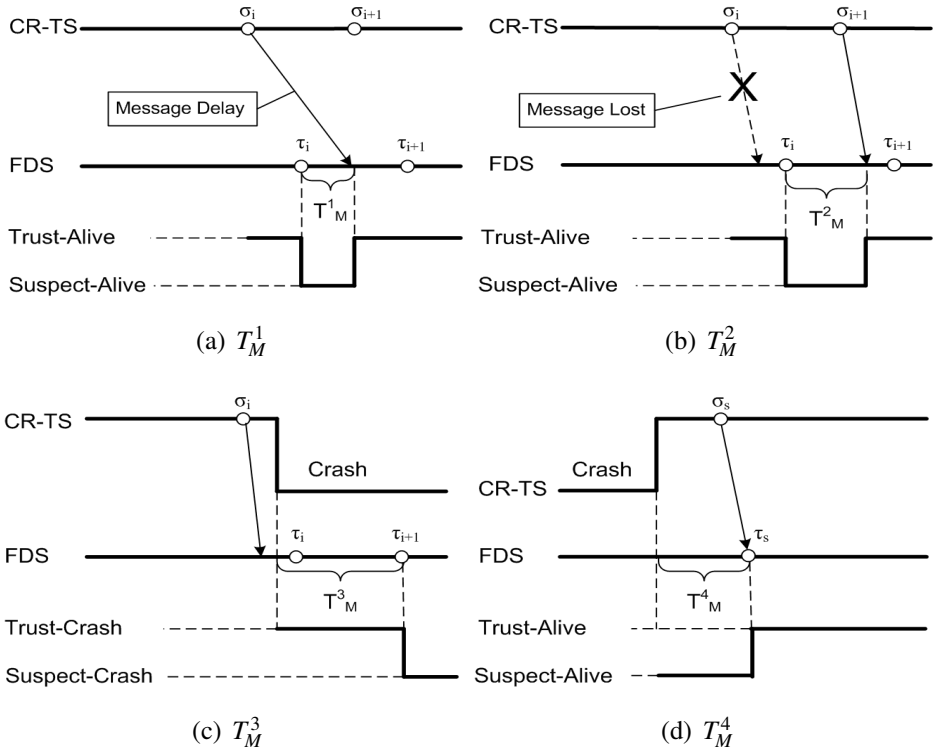


Figure 5.11: The Analysis of Possible T_M in a Crash-Recovery Run

From the above analysis we know that for the same η, p_L, $E(D)$, MTTF and MTTR, when the *timeout* length increases, the average mistake duration caused by message delays and the average mistake duration caused by message losses will decrease ($T_M^1 \downarrow$ and $T_M^2 \downarrow$), the average mistake duration caused by the CR-TS's crash will increase ($T_M^3 \uparrow$), and the average mistake caused by the CR-TS's recovery from a detectable crash is not affected by the *timeout* length ($\overline{T_M^4}$) but fewer crashes and recoveries will be detected. In the simulation cases 1 and 2 p_L=0.01 and MTBF = 105, when *timeout* is small, T_M^2 and T_M^3 occur with similar frequency. When *timeout* increases from 1.5 to 2.0, (the

FDS can tolerate zero message loss and most message delays), $E(T_M)$ increases slow because T_M^1 ⌊, T_M^2 ⌊, T_M^3 ⌈ and $\overline{T_M^4}$ and their impacts counterbalance. Overall $E(T_M)$ is stable within this period. As the *timeout* length increases, T_M^2 will occur less frequently. But T_M^3 occurs every MTBF period. Thus, as the *timeout* increases, T_M^3 will become dominant and $E(T_M)$ will increase gradually.

In the simulation cases 3 and 4, p_L=0.01 and MTBF = 1050. When the *timeout* length is small, T_M^2 will have more impact than T_M^3, because T_M^2 occurs more frequently than the crash and recovery. Therefore, as the *timeout* length increases, the average duration of T_M^2 decreases and T_M^2 occurs less frequently; $E(T_M)$ will increase slower or even decrease since more message losses are tolerated. But if *timeout* continually increases, T_M^3 will become dominant, and $E(T_M)$ will later increase gradually.

Overall, Fig. 5.10 shows that in a *crash-recovery* run, $E(T_M)$ exhibits quite different characteristics from a *fail-free* run. If the message delay and the probability of message loss are not very large, $E(T_M)$ is bounded by MTTR. From Fig. 5.10, we also observe that $E(T_M)$ can possibly be decreased when some *timeout* value is chosen. In a *crash-recovery* run, continually increasing the *timeout* length cannot achieve a better (T_M) as in a *fail-free* run.

Fig. 5.12 demonstrates $E(T_{MR})$ of the simple timeout algorithm with exponential or deterministic MTTF and MTTR with various values. We can see that as MTBF increases, for the same *timeout* length, $E(T_{MR})$ increases as well. This implies that $E(T_{MR})$ is greatly impacted by the dependability of the CR-TS. We can also see that, for all of these four simulation cases, $E(T_{MR})$ increases exponentially fast at the beginning but after $E(T_{MR})$ reaches $\frac{\text{MTBF}}{2}$, it will stop increasing exponentially. For the CR-TS with deterministic MTTR, $E(T_{MR})$ will stop at $\frac{\text{MTBF}}{2}$ when failures are detectable. This is because for the deterministic MTTR, when the *timeout* length is smaller than MTTR, all crashes are still detectable. Even if all of the message delays and losses are tolerated, for every MTBF period there are still two mistakes (T_M^3, T_M^4) which will certainly occur. Thus $E(T_{MR}) \leq \frac{\text{MTBF}}{2}$ within this duration (see Lemma 3.6.5). If T_D is larger than the recovery duration, all crashes might become undetectable (see the proof of Lemma 3.6.9). When more and more message delays and losses are tolerated, $E(T_{MR})$ will become stable at MTBF (see Lemma 3.6.4). For the CR-TS with exponential

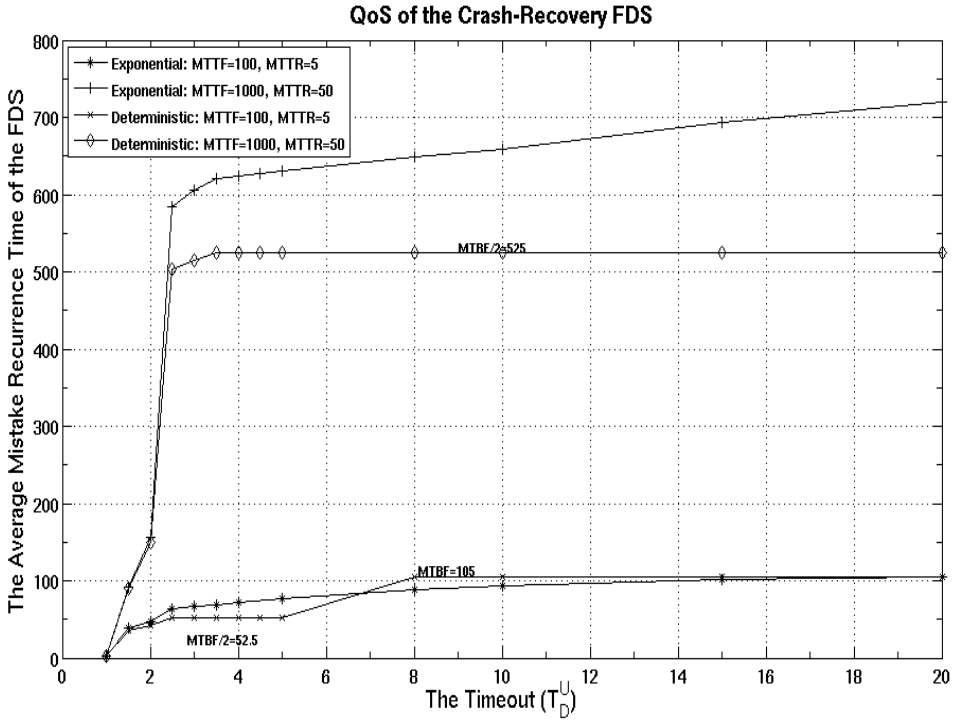


Figure 5.12: The Simple Timeout Algorithm: $E(T_{MR})$

MTTR, $E(T_{MR})$ will increase gradually and approach MTBF rather than stopping at $\frac{\text{MTBF}}{2}$, until all crashes become undetectable. This is because for non-deterministic MTTR, as the *timeout* length increases, the proportion of the detectable crashes decreases. Therefore, for the detectable crashes $T_{MR} \leq \frac{\text{MTBF}}{2}$ and for the undetectable crashes $T_{MR} \leq \text{MTBF}$. Thus $E(T_{MR})$ will increase gradually between $[\frac{\text{MTBF}}{2}, \text{MTBF}]$ and finally stabilize at MTBF. All of these results match our analysis in Chapter 3 well and indicate that if a CR-TS is not *fail-free* (MTTF $\rightarrow \infty$) or *crash-stop* (MTTR $\rightarrow \infty$), $E(T_{MR})$ will be bounded by MTBF when failures are undetectable and $\frac{\text{MTBF}}{2}$ when failures are detectable.

Fig. 5.13 shows for the same QoS of message communication, when MTBF increases, P_A will be improved. This is because as MTBF increases, $E(T_{MR})$ increases as well. Thus from the equation $P_A = 1 - \frac{E(T_M)}{E(T_{MR})}$ (as in [24]), we can know that for the same *timeout* length, when MTBF increases, a better P_A can be achieved. However, from

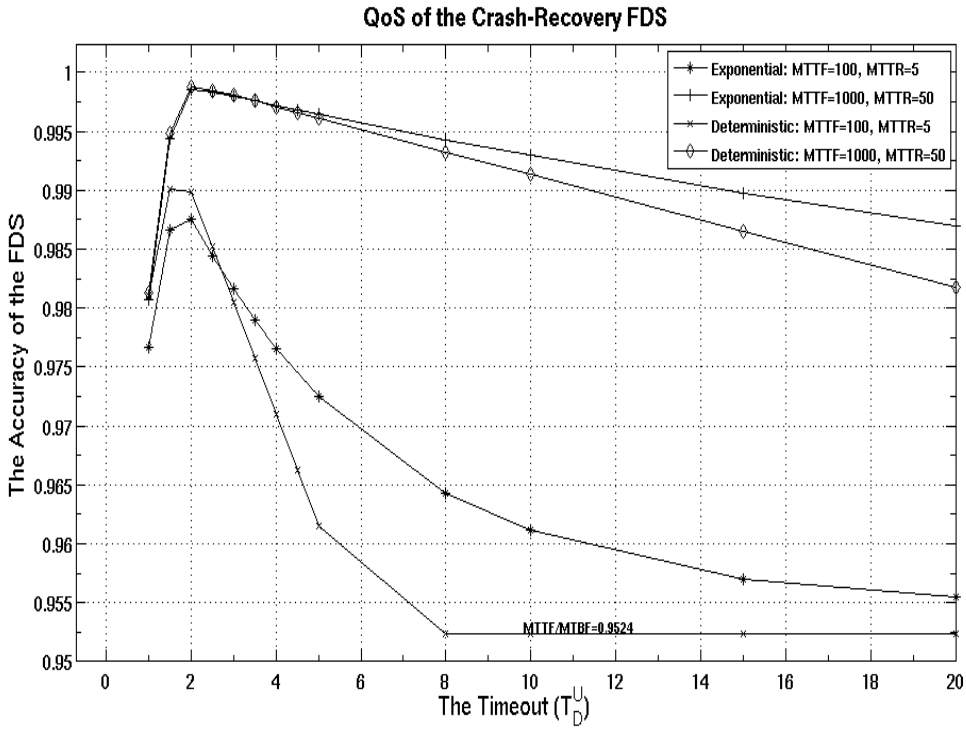


Figure 5.13: The Simple Timeout Algorithm: P_A

the Fig. 5.13, we can also see that as the *timeout* length increases, P_A is not always increasing as in a *fail-free* or *crash-stop* run. Continually increasing *timeout* could decrease P_A. This is because T_{MR} is bounded by $\frac{\text{MTBF}}{2}$ or MTBF. After $E(T_{MR})$ reaches $\frac{\text{MTBF}}{2}$, it increases slowly rather than exponentially fast but $E(T_M)$ increases linearly and faster than $E(T_{MR})$. Thus P_A decreases and finally P_A will approach $\frac{\text{MTTF}}{\text{MTBF}}$, which equals the availability of the CR-TS.

All of the above results indicate that for a highly available CR-TS, even if the FDS always trusts the CR-TS without detecting a crash, in terms of the QoS metrics in [24], a quite good QoS of the FDS can still be achieved. Especially for a highly available and highly consistent but not highly reliable CR-TS, this QoS (as in [24]) might be the best one for such a CR-TS. But the *completeness* property (see Section 3.6.3) of the FDS is not satisfied, since no failure is detected. Consequently, these simulation results demonstrate the necessity of the additional QoS metrics we proposed in Section 3.6.3

to measure the *completeness* aspects and the speed of the recovery detection of a *crash-recovery* FDS. Furthermore, these results also demonstrate the necessity of adopting the recovery detection protocols we presented in Chapter 4, which can improve the proportion of detected failures without reducing other QoS aspects. We will introduce the simulation results of how well the NFD-S algorithm with the lightweight recovery detection protocol can improve the QoS of the FDS in the following sections.

5.4.2 Evaluation of the NFD-S and NFD-S-LRD Algorithms

5.4.2.1 Analysis for the Basic QoS Metrics

We implemented the NFD-S algorithm presented in [24] and the NFD-S-LRD algorithm (see Section 4.3.2) to evaluate the QoS of both algorithms and compared them with the analytical results derived from Theorem 3.6.2 (see simulation cases 5-8 in Table 5.1). Figs. 5.14-5.19 compare the QoS of the two algorithms (simulation results) and the corresponding analytical results from different perspectives. For the analytical results, in a *crash-recovery* run, some of the inequalities (such as T_M, T_{MR}, P_A) in Theorem 3.6.2 are related to $Pr(X_a > \tau_i - t_r)$ (see Proposition 3.6.1). Therefore the QoS bound estimation will be changing dynamically as $Pr(X_a > \tau_i - t_r)$ is changing according to the current CR-TS's alive duration. This will bring more difficulty to the QoS estimation. However, since we only estimate the bounds of each QoS metric, we can maximize $Pr(X_a > \tau_i - t_r)$, which can be done by setting the heartbeat sequence number i to be a small number. This is because when the maximum $Pr(X_a > \tau_i - t_r)$ is adopted, p_s^i will be maximized, which means the maximum mistake number and mistake duration within the *fail-free* duration will be obtained. (See the proof of Proposition 3.6.1 and Fig. 3.7). Thus we can estimate the upper bound of the average mistake duration ($E(T_M^U)$) and the lower bound of average mistake recurrence time ($E(T_{MR}^L)$) and the lower bound query accuracy probability (P_A^L). In our analytical results computation, we set the heartbeat sequence number $i = 1$. Note that if a large i is adopted, the analytical results might still be valid and closer to the simulation results. However, the possibility that the average number and duration of the mistakes within the *fail-free* duration are underestimated will increase, which means that the QoS of the FDS might

not satisfy the given QoS requirements.

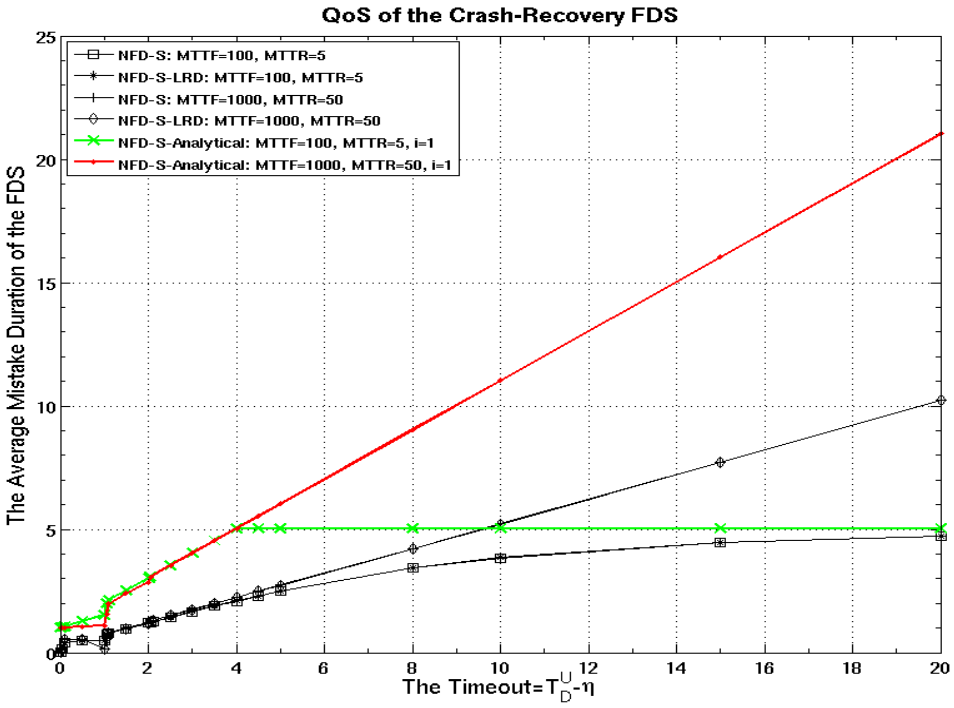


Figure 5.14: The NFD-S and NFD-S-LRD Algorithms: $E(T_M)$

Figs. 5.14-5.16 show the comparison in terms of $E(T_M)$, $E(T_{MR})$ and P_A, which are the QoS metrics proposed in [24]. From these three figures, we have the following observations:

1. With the same dependability and the same QoS of message transmission, the QoS of the NFD-S algorithm and the NFD-S-LRD algorithm are quite similar to the simple timeout algorithm. The NFD-S and NFD-S-LRD algorithms perform slightly better than the simple one. Since the detailed analysis of the shapes of the QoS metrics is presented in Section 5.4.1, we do not repeat that analysis here. As we can observe from Figs. 5.10, Fig. 5.12, Fig. 5.13 and Figs. 5.14-5.16, the dependability of a CR-TS can influence the QoS of the FDS. Particularly, for a highly available but not highly reliable CR-TS, the dependability of the CR-TS can have more impact than the performance of the algorithm and the QoS of

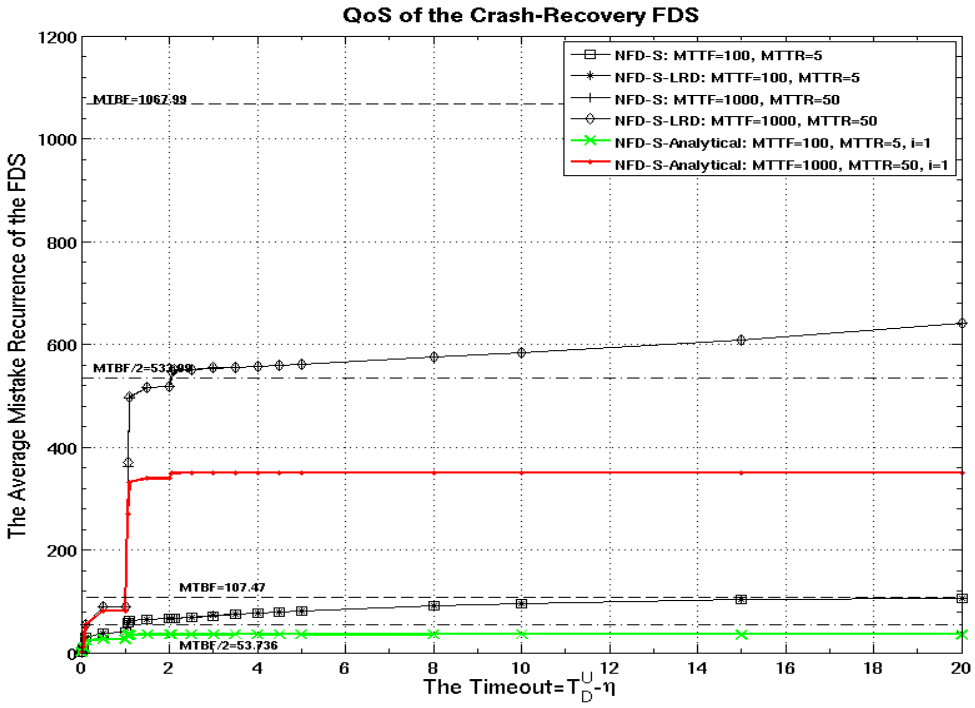


Figure 5.15: The NFD-S and NFD-S-LRD Algorithms: $E(T_{MR})$

message transmission. In such situations, the dependability of the CR-TS must be taken into account for the FDS design and implementation.

2. With the same dependability and QoS of message transmission, the NFD-S algorithm and the NFD-S-LRD algorithm perform equally well in terms of $E(T_M)$, $E(T_{MR})$, P_A. This is because the lightweight recovery detection protocol is designed for improving the *completeness* (R_{DF} and R_{DR}) without affecting the other QoS aspects of the FDS. Based on Figs. 5.14-5.16, we can conclude that the NFD-S-LRD algorithm achieves our goals.

3. From Figs. 5.10, Fig. 5.12, Fig. 5.13 and Figs. 5.14-5.16, we can see that P_A, $E(T_{MR})$ and $E(T_M)$ have bounds. Continually increasing the *timeout* length might not be a reasonable way to achieve better P_A, $E(T_{MR})$ and $E(T_M)$. A potential trade-off exists between the QoS metrics. For instance, for both the NFD-S algorithm and the NFD-S-LRD algorithm, $timeout \in [1,\ 1.1]$ ($T_D^U =$

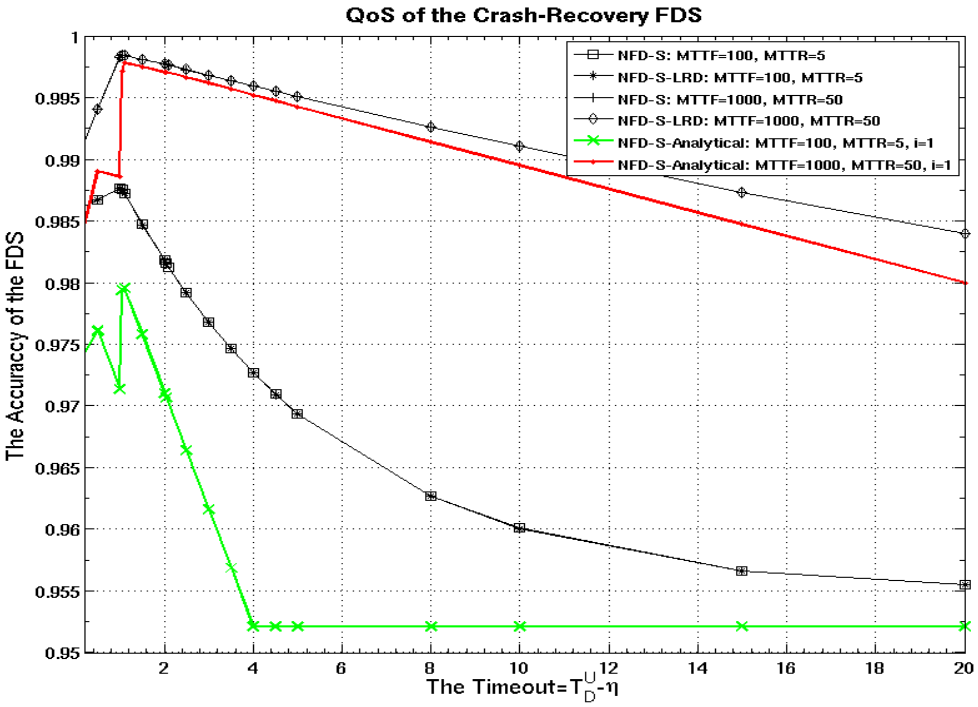


Figure 5.16: The NFD-S and NFD-S-LRD Algorithms: P_A

$timeout + \eta \in [2,\ 2.1]$) might achieve the best overall QoS. For the simple timeout algorithm $timeout \in (2,\ 2.1)$ ($T_D^U \in [2,\ 2.1]$) might achieve the best over all QoS.

4. Compared with a *fail-free* or *crash-stop* run, $E(T_M)$ in a *crash-recovery* run exhibits quite different characteristics. This is because in a *crash-recovery* run, the mistakes caused by the crash and recovery are taken into consideration, which means continually increasing the *timeout* length will not always decrease $E(T_M)$ as in a *fail-free* or a *crash-stop* run. It is possible to increase *false positive* mistakes (T_M^3, see Fig. 5.11(c)) as well. As the *timeout* length increases, mistakes caused by message delays and losses will occur less frequently, thus *false positive* mistakes will start to dominate the QoS of the FDS, which statistically have not been involved in the previous work focusing on the QoS of the *crash-stop* failure detectors.

5. From Figs. 5.14-5.16, we can observe that, the simulation results of $E(T_M)$ are smaller than the analytical results, the simulation results of $E(T_{MR})$ are larger than the analytical results and the simulation results of P_A are larger than the analytical results, which indicate that the bound analysis of the basic QoS metrics in Theorem 3.6.2 are valid and the simulation results satisfy the QoS requirements according to the analysis.

6. From Figs. 5.14-5.16, we can also observe an obvious gap between the analytical results and the simulation results. This is caused by the overestimated or underestimated analytical results within a *crash-recovery* run but the simulation results are the statistics of the average results. Recall that in the proof of Lemma 3.6.6, $E(T_M)$ is overestimated as an upper bound by using the total mistake duration over the underestimated average number of mistakes that might occur within a *crash-recovery* duration. Thus the analytical results of $E(T_M)$ will be larger than the simulation results. For $E(T_{MR})$, in the proof of Lemma 3.6.3, it is estimated as a lower bound by using the observation duration (MTBF) over an overestimated number of mistakes that might occur within a *crash-recovery* duration. For instance, the number of mistakes within the crash duration is estimated as $\lceil \frac{E(D)}{\eta} \rceil + 1$ (see the proof of Lemma 3.6.2), which is an upper bound rather than the average number. Thus if such an estimation is adopted, $E(T_{MR})$ of the analytical results will be smaller than the simulation results. For P_A, in the proof of Lemma 3.6.5, it is estimated as a lower bound by using one minus an overestimated total mistake duration over the observation duration (MTBF). Thus P_A of the analytical results will be smaller than the simulation results. All of these results satisfy the QoS requirements $E(T_M) < T_M^U$, $P_A > P_A^L$ and $E(T_{MR}) > T_{MR}^L$. Notice that if the analytical results can be estimated closer to the actual average results, the analytical results will approach to the simulation results more closely as well.

5.4.2.2 Analysis for the Extended QoS Metrics

In order to compare the differences between the NFD-S and NFD-S-LRD algorithms, we also plot $E(R_{DF})$, $E(T_D)$ and $E(T_{DR})$ in Figs. 5.17-5.19 to demonstrate the performance comparison.

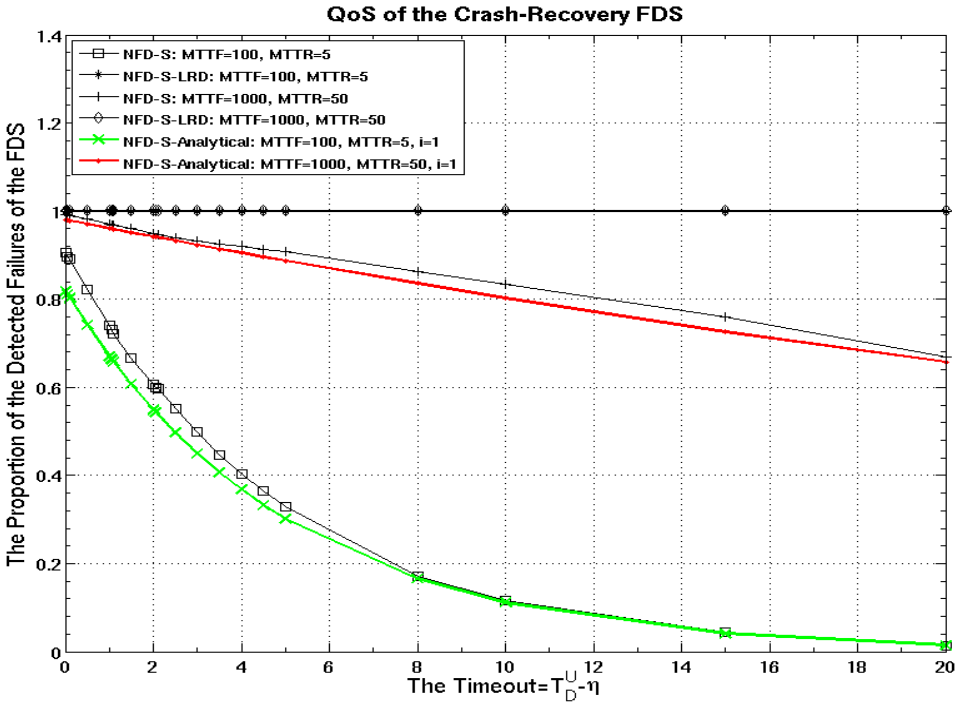


Figure 5.17: The NFD-S and NFD-S-LRD Algorithms: $E(R_{DF})$

Fig. 5.17 shows the proportion of the detected failures of the NFD-S and NFD-S-LRD algorithm with different dependability of the CR-TS for both simulation and analytical results. Recall that P_A is the probability of the FDS which can output an accurate result at an arbitrary time whereas R_{DF} is the proportion of the detected failures among the occurred failures (see the definition of P_A in Section 3.6.2 and the definition of R_{DF} in Section 3.6.3). Therefore R_{DF} is differ from P_A. As the *timeout* length increases, $E(R_{DF})$ of the NFD-S algorithm decreases. When MTTR becomes shorter, $E(R_{DF})$ will decrease faster. This is because the smaller MTTR is, the faster $timeout + \eta$

crosses MTTR ($T_D^U >$ MTTR). Therefore, more crashes remain undetected when the NFD-S algorithm is adopted. From the Fig. 5.17, we can also see that the simulation results of $E(R_{DF})$ are larger than the analytical results, which means the bound analysis of $E(R_{DF})$ is valid and the simulation results satisfy the QoS requirements in terms of R_{DF}^L. In addition, for the NFD-S-LRD algorithm, from Fig 5.17 we can see that $E(R_{DF})$ is not affected by the length of *timeout* and MTTR. $E(R_{DF})$ of the NFD-S-LRD algorithm remains stable and approaches 1.0, which means almost all crashes are detected. Thus we can conclude that in terms of the *completeness* property, the NFD-S-LRD algorithm performs better than the NFD-S algorithm in a *crash-recovery* run. From Fig. 5.17, we can also see a gap between the analytical results and the simulation results. This is caused by the worst case estimation. In the proof of Lemma 3.6.9, we assume the CR-TS crashes just after a liveness message is sent as the worst case and in order to detect an occurred crash failure, X_c greater than $timeout + \eta$ is needed (see Fig. 3.8). However, many failures do not occur just after a liveness message is sent. Therefore the proportion of the detected failure is underestimated and the analytical results are smaller than the simulation results, which satisfies the QoS requirement $E(R_{DF}) > R_{DF}^L$.

From Fig. 5.18, we can observe that with the same QoS of message communication and the CR-TS's dependability, $E(T_D)$ of the NFD-S-LRD algorithm is smaller than the NFD-S algorithm and the upper bound of $E(T_D)$ of the NFD-S-LRD algorithm is MTTR $+ E(T_{DR})$. This is because when a recovery detection protocol is adopted, after the recovery of the CR-TS, the FDS takes $E(T_{DR})$ time units to detect the recovery on average, which means that from the crash time, the FDS takes MTTR $+ E(T_{DR})$ time units to detect the occurred crash. This result shows that the recovery detection protocol could be particularly useful to detect a crash failure for a highly consistent CR-TS. Since all of the existing failure detection algorithms adopt increasing the *timeout* length to tolerate more message losses and delays, if a CR-TS is recoverable and recovers fast, it could be difficult for these algorithms to achieve the QoS in [24] and satisfy the *completeness* property at the same time. By adopting the recovery detection protocol, this problem can be solved reasonably well. In addition, according to the NFD-S algorithm, T_D is bounded by $\eta + timeout$ regardless the correctness of the detection, thus $T_D < T_D^U$ must be satisfied. In Fig. 5.18, we do not plot $\eta + timeout$

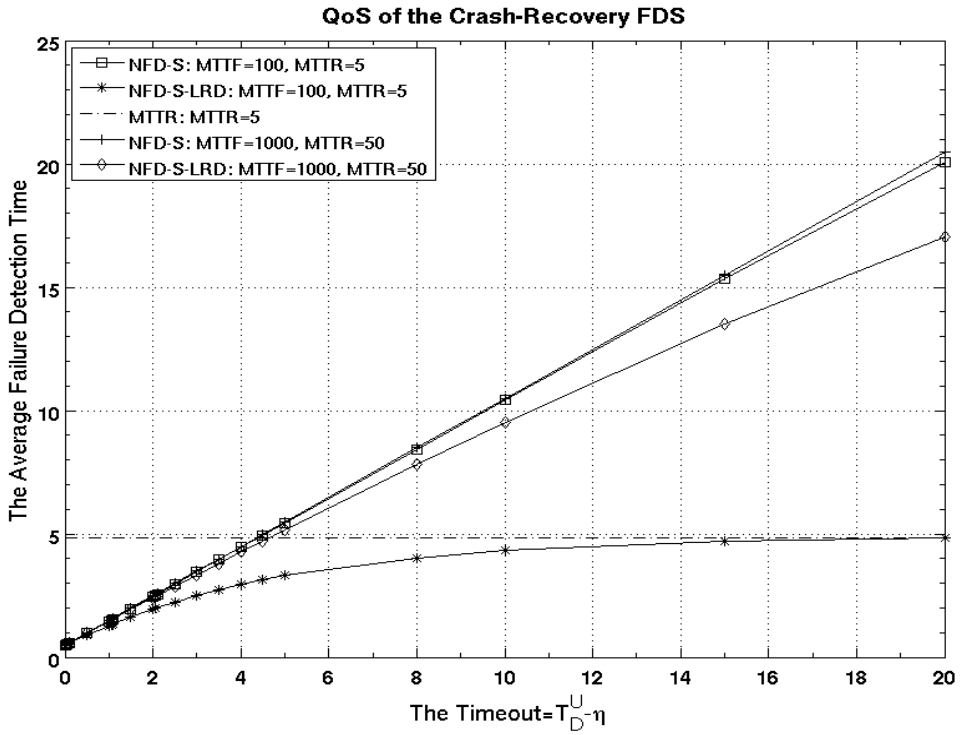


Figure 5.18: The NFD-S and NFD-S-LRD Algorithms: $E(T_D)$

which can be observed directly from the horizonal axis.

Fig. 5.19 shows the $E(T_{DR})$ of the NFD-S-LRD algorithm with the same QoS for the message communications. We observe that $E(T_{DR})$ is not affected by MTTF, MTTR and the *timeout* length. It stays stable around 0.03 (see Fig. 5.19)[5]. According to Lemma 4.2.1 and Remark 4.2.1 in Chapter 4, we know that T_{DR} is only affected by the heartbeat interval η, the consecutive message loss length X_L and the message delay D. Thus $E(T_{DR})$ can be estimated by $E(T_{DR}) = \eta \cdot E(X_L) + E(D)$. Since each message's transmission in our simulation is independent and theoretically the possible consecutive message loss number could be $+\infty$. From Remark 4.2.1, we can get $E(T_{DR}) = \eta \times \frac{p_L}{1-p_L} + E(D) \approx 0.03$. Therefore, the simulation results demonstrate the Lemma 4.2.1 and Remark 4.2.1 quite well. In addition, Figs. 5.17-5.19 also demon-

[5]The bias between the two simulation results is caused by the different simulation length for each simulation (see Section 5.3), which will result in a little different p_L, $E(D)$ and $E(X_L)$

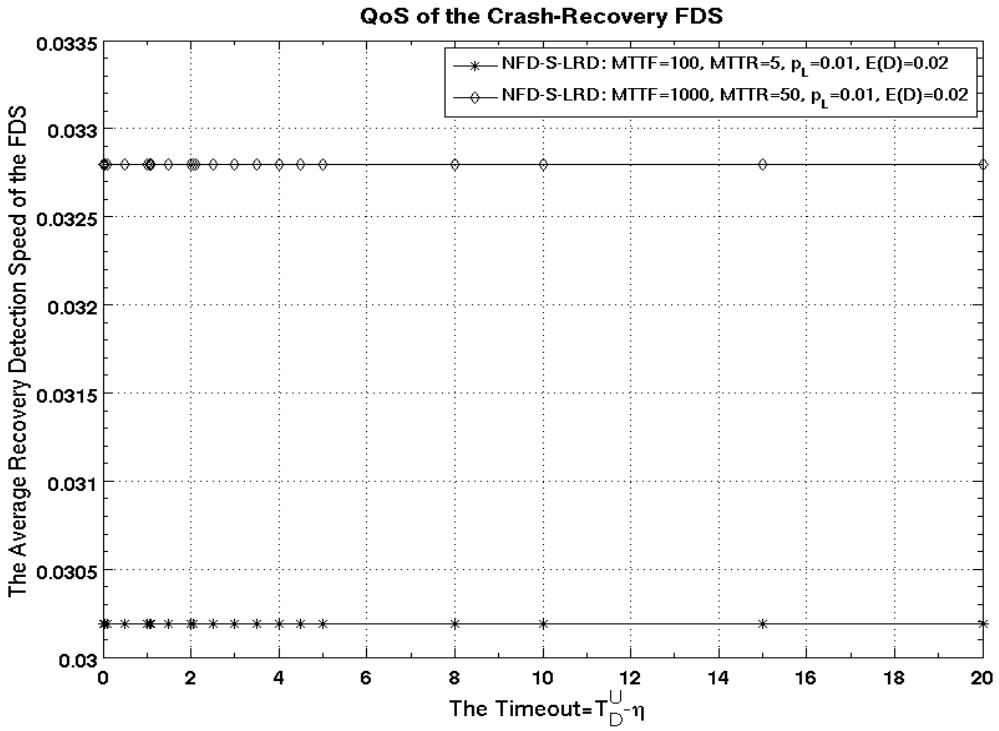


Figure 5.19: The NFD-S and NFD-S-LRD Algorithms: $E(T_{DR})$

strate the lightweight recovery detection protocol can adapt to the *crash-recovery* behavior of the CR-TS well. It also allows the FDS to have a consistent view of the CR-TS and let the FDS know about all state changes of the CR-TS. This means that recovery detection protocols can improve the adaptivity of a FDS and enable it operate autonomously. Especially for the highly consistent *crash-recovery* CR-TS, this feature is particularly useful.

5.5 Conclusions

In this Chapter, we introduced the design and implementation of our FD-Simulator, which aims to simplify complex failure detection frameworks and algorithms simulation. The FD-Simulator can construct the simulation framework automatically accord-

ing to the XML-based configuration file and gather all of the QoS performance data in a file after the simulation for further analysis.

By using the FD-Simulator, we also designed some simulation cases directed to assessing the QoS of the *crash-recovery* FDS and the lightweight recovery detection protocol based on the NFD-S algorithm. We compared the simulation results with the derived analytical results and analyzed them in detail. Our analysis in Section 5.4 shows that due to the *crash-recovery* of a CR-TS, a FDS cannot achieve 100% accuracy. The dependability of a recoverable monitored target can impact the QoS of failure detectors. For a *crash-recovery* FDS, $E(T_{MR})$ is bounded by MTBF, P_A is bounded by $\frac{\text{MTTF}}{\text{MTBF}}$ and $E(T_M)$ is bounded by MTTR. All of the simulation results imply that when a failure detector is designed and implemented, the dependability of the CR-TS needs to be considered. In addition, the plotted Figures in Section 5.4.2 demonstrate that our QoS bound analysis in Chapter 3 is valid and can be used as an approximate solution for the computation of FDS's parameters or the QoS bounds estimation if the FDS's parameters are given. The simulation results also show that the NFD-S-LRD algorithm could be a good solution for a highly consistent monitored target with reasonable MTTF length. Compared with the NFD-S algorithm, the NFD-S-LRD algorithm exhibits a better QoS in terms of the *completeness* of a *crash-recovery* failure detector, and performs equally well for other QoS aspects. The NFD-S-LRD algorithm is more flexible and adaptive to the highly dynamic distributed system and satisfies the autonomy requirement (see Section 2.7) of a failure detection task. Especially within a large-scale distributed network, self-management of a failure detection framework, which can adapt to the monitored target's *crash-recovery* behavior can reduce the complexity of monitoring procedures.

Finally, the recovery protocols we proposed should also work well with other failure detection algorithms. The NFD-S is just one possible option. Other algorithms can certainly be combined with a recovery detection protocol to achieve a better QoS from different perspectives.

Chapter 6

Conclusions and Future Directions

6.1 Introduction

In this chapter, the main results of this thesis are summarized in Section 6.2. Some possible future directions are introduced in Section 6.3.

6.2 Conclusions

Dependability is an important issue for any applications or systems. Various attributes are proposed to measure the system dependability and different fault-tolerance mechanisms are adopted to achieve such a goal. Failure detectors are critical building blocks for fault-tolerant computing. In this thesis, crash failure detection has been investigated as the most fundamental problem. Previously, the study of crash failure detectors and their QoS has been based on the *crash-stop* or *fail-free* assumption, only considering the impact of message transmission. The impact of the system dependability on the QoS of crash failure detectors is not considered. In this thesis, we have shown that the *crash-stop* run and *fail-free* run are particular cases of the *crash-recovery* run and that dependability measurements, such as reliability, availability, consistency, should be involved for the QoS analysis of the *crash-recovery* failure detection. We have extended the existing QoS metrics of *crash-stop* failure detectors to adapt to the *crash-recovery*

behaviour of a monitored target and shown how to configure a failure detector in a *crash-recovery* run. We analyzed the approximate QoS bounds in depth, showed the configuration procedure to satisfy a given set of QoS requirements and discussed how to estimate the failure detector's input parameters in *crash-recovery* runs. In addition, two types of recovery detection protocol were proposed to enable the failure detector to be more adaptive to the recovery of the monitored target and improve the *completeness* of the QoS of failure detectors. The first protocol is the recovery detection protocol with persistent storage, which can guarantee the detection of each crash failure and recovery, estimate each recovery time and satisfy the *strong completeness* requirement. The second one is the lightweight recovery detection protocol without using persistent storage, which is designed to reduce the system overhead and can detect most recoveries of the monitored target. Both of these two protocols can improve the QoS of failure detectors and enable the *crash-recovery* failure detection as an autonomous monitoring procedure.

Furthermore, in order to evaluate the QoS of a failure detector in a *crash-recovery* run, we have developed the FD-Simulator, which can construct a failure detection framework according to an XML-based configuration file. By using the FD-Simulator, the simple timeout algorithm, the NFD-S algorithm and the NFD-S algorithm with the lightweight recovery detection protocol have been assessed with various MTTF and MTTR. The simulation results have been analyzed in detail and show that in a *crash-recovery* run, the QoS of failure detectors exhibit quite different characteristics compared with a *crash-stop* or *fail-free* run, which matches our analysis in Chapter 3. The simulation results also demonstrate that the dependability of the monitored target has significant impact on the QoS of crash failure detectors, particularly when the monitored target is not reliable and highly consistent. Finally, we showed that the proposed lightweight recovery detection protocol works well when the monitored target has a reasonable alive time and it can improve the proportion of the detected failures and recoveries.

Overall, this thesis is only one step towards solving failure detection problems for distributed systems. More investigations are certainly needed in future research and some possible future directions are given in the following section.

6.3 Future Directions

In this thesis, we focus on two-process systems: one failure detector and one monitored target. In practice, there might be one failure detector which monitors many targets, or many failure detectors monitoring many targets. In such cases, introducing redundancy into the system to avoid a single point failure of the failure detector and analyzing the dependability of the failure detection system are interesting topics. For such systems, adopting multicast, group membership and gossip protocols to provide probabilistic QoS coverage and balance the system overhead could be a challenging problem for more scalable failure detectors. Particularly for a hierarchical failure detection framework, providing a QoS guarantee and balancing the system overhead could be a potential direction for future research.

For both of flat and hierarchical failure detection systems, designing communication protocols to reduce the communication overhead and promote the QoS of communication will be an interesting topic to investigate. Topics to consider include the trade-off between the QoS of failure detectors and the system overhead by adopting unreliable lightweight communication protocols or reliable heavyweight communication protocols; or designing local level communication protocols or global level communication protocols in hierarchical failure detection systems.

Recent statistical work [75] shows that probabilistic failure behaviours within distributed systems are complicated. The exponential distribution does not faithfully represent the failure behaviours. MTTF depends more on the size of the system and MTTR depends more on the type of the system. Therefore, involving more failure information and adopting more complicated failure analysis mechanisms to improve the QoS of failure detectors is a promising direction for the future failure detection study. From long term monitoring, system failure and recovery behaviours could be trackable more accurately. Thus, collecting and analyzing this failure information using statistical methods such as Bayesian machine learning to improve the QoS of failure detection can be a possible way to analyze the failure detector and the monitored target more precisely.

For any failure detection system, the performance of the network will greatly influ-

ence the QoS of failure detectors. Sometimes, network traffic might be bursty (as in [81]). For example, the message delay and loss rate might be higher during peak times. Adopting more complicated network performance analysis and forecasting mechanisms, such as using Markov models or Bayesian methods to predict the message transmission more accurately, is still an important issue.

For service-oriented failure detection, any failure might cause a service composition failure. Considering more general failure modes such as QoS failure in Section 2.6 and using verification mechanisms to identify such failures could be particularly helpful in a service-oriented environment. In addition, dependencies exist between many components within distributed systems. For such systems, failures will have chained behaviours. For instance, a component failure might cause a composed service to fail as well; a memory leak or a hard disk failure will cause a machine failure and such machine failure will cause service failure. Therefore injecting many different types of sensor could possibly improve the accuracy of failure detectors and reduce the costs of dealing with occurred failures and recoveries. For example, if the service A fails and keeps silence caused by the crash of the service B, it would be better to recover both of them rather than just recover the service A; if both the hosting environment and the hosted service have crashed, it will be useless to try service replications on the same host machine. For a system which requires high availability, such diagnosis could be particularly useful. Overall, tracking the root cause of a service failure and figuring out the dependency between different components can be implemented by multiple types of sensor deployment and fault-tree based failure diagnosis, which are interesting topics for future study.

Some recent research work in [5, 72] shows that probabilistic checkpointing, rejuvenation and recovery can achieve better performance and improve the application's reliability and availability. Combining probabilistic failure analysis, failure detection with probabilistic checkpointing, rejuvenation and recovery mechanism could be a way to achieve better performance and balance the overhead of checkpointing and recovery, which might be a direction worth investigating.

Bibliography

[1] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure Detection and Consensus in the Crash-Recovery Model. *Distributed Computing*, 13(2):99 – 125, Apr. 2000.

[2] Marcos Kawazoe Aguilera, Sam Toueg, and Borislav Deianov. On the Weakest Failure Detector for Uniform Reliable Broadcast. Technical Report TR99 - 1741, Department of Computer Science, Upson Hall, Cornell University, 30 1999.

[3] Carlos Almeida and Paulo Verissimo. Timing Failure Detection and Real-Time Group Communication in Quasi-Synchronous Systems. In *Proceedings of the Eighth Euromicro Workshop on Real-Time Systems*, pages 230 – 235, Los Alamitos, CA, USA, Jun. 1996. IEEE Computer Society.

[4] Stuart Anderson, Yin Chen, Glen Dobson, Stephen Hall, Conrad Hughes, Yong Li, Sheng Qu, Ed Smith, Ian Sommerville, and Tiejun Ma. Dependable Grid Services. In *UK e-Science All Hands Meeting*, 2003. Nottingham, UK.

[5] Alberto Avritzer, Andre Bondi, Michael Grottke, Kishor S.Trivedi, and Elaine J.Weyuker. Performance Assurance via Software Rejuvenation: Monitoring, Statistics and Algorithms. In *Proceedings of the 2006 International Conference on Dependable Systems and Networks(DSN) Philadelphia*, volume 0, pages 435 – 444, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[6] Lokesh Bajaj, Mineo Takai, Rajat Ahuja, Ken Tang, Rajive Bagrodia, and Mario Gerla. GloMoSim: A Scalable Network Simulation Environment. *UCLA Computer Science Department Technical Report*, 990027, 1999.

[7] Keith Ballinger, Peter Brittenham, Ashok Malhotra, William A. Nagy, and Stefan Pharies. Web Services Inspection Language (WS-Inspection) 1.0. `http://www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html`, Nov. 2001.

[8] William H. Bell, David G. Cameron, A. Paul Millar, Luigi Capozza, Kurt Stockinger, and Floriano Zini. Optorsim: A Grid Simulator for Studying Dynamic Data Replication Strategies. *International Journal of High Performance Computing Applications*, 17(4):403 – 416, 2003.

[9] M. Bertier, O. Marin, and P. Sens. Implementation and Performance Evaluation of an Adaptable Failure Detector. In *Proceedings. International Conference on Dependable Systems and Networks. Washington D.C., USA*, pages 354 – 363, 2002.

[10] Ranjita Bhagwan, Stefan Savage, and Geoffrey M. Voelker. Replication Strategies for Highly Available Peer-to-peer Storage Systems. In *Proceedings of FuDiCo: Future directions in Distributed Computing*, Jun. 2002.

[11] Kenneth P. Birman. Replication and Fault-Tolerance in the ISIS System. In *Proceedings of the tenth ACM Symposium on Operating Systems Principles*, volume 19(5), pages 79 – 86. ACM Press New York, NY, USA, 1985.

[12] G.S. Blair, G. Coulson, L. Blair, H. Duran-Limon, P. Grace, R. Moreira, and N. Parlavantzas. Reflection, Self-awareness and Self-Healing in OpenORB. In *Proceedings of the First Workshop on Self-Healing Systems*, pages 9–14. ACM Press New York, NY, USA, 2002.

[13] A. Bondavalli and L. Simoncini. Failure Classification with Respect to Detection. In *Proceedings of Second IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 47 – 53. Cairo, Egypt, 30 Sep. - 2 Oct. 1990.

[14] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. Web Services Architecture. `http://www.w3.org/TR/ws-arch/`, Feb. 2004.

[15] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendel-

sohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol (SOAP) 1.1. `http://www.w3.org/TR/2000/NOTE-SOAP-20000508`, May 2000.

[16] Rajkumar Buyya and Manzur Murshed. GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing. *Concurrency and Computation: Practice and Experience*, 14(13 - 15):1175 – 1220, 2002.

[17] George Candea and Armando Fox. Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel. In *HOTOS '01:Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, pages 125–132, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[18] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot-A Technique for Cheap Recovery. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–44, Dec. 2004.

[19] Henri Casanova. Simgrid: A Toolkit for the Simulation of Application Scheduling. In *Proceedings of the IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 430, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[20] Antonio Casimiro and Paulo Verissimo. Timing Failure Detection with a Timely Computing Base. In *Proceedings of the European Research Seminar on Advances in Distributed Systems (ERSADS'99)*, Apr. 23 - 28 1999.

[21] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The Weakest Failure Detector for Solving Consensus. In *Procceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing (PODC'92)*, pages 147 – 158, Vancouver, BC, Canada, 1992. ACM Press.

[22] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for asynchronous systems (preliminary version). In *PODC '91: Proceedings of the tenth*

annual ACM symposium on Principles of distributed computing, pages 325–340, New York, NY, USA, 1991. ACM Press.

[23] Tushar Deepak Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225 – 267, 1996.

[24] Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera. On the Quality of Service of Failure Detectors. *IEEE Transactions on Computers*, 51(5):561 – 580, 2002.

[25] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`, 15 March 2001.

[26] John C.Knight and Elishabeth A.Strunk. Software Dependability. Int. Conf. on Dependable Systems and Networks, Tutorials, June 2006.

[27] James Clark and Steve DeRose. XML Path Language (XPath) Version 1.0. `http://www.w3.org/TR/xpath`, 1999.

[28] Luc Clement, Andrew Hately, Claus von Riegen, and Tony Rogers. UDDI Version 3.0.2 Specification. `http://uddi.org/pubs/uddi-v3.0.2-20041019.htm`, Oct. 2004.

[29] John N. Daigle. *Queueing Theory with Applications to Packet Telecommunication*. Springer, Nov. 30 2004.

[30] David Daly, Daniel D. Deavours, Jay M. Doyle, Patrick G. Webster, and William H. Sanders. Möbius: An Extensible Tool for Performance and Dependability Modeling. *11th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS), Schaumburg, IL. Lecture Notes in Computer Science*, 1786:332 – 336, March 2000.

[31] Abhinandan Das, Indranil Gupta, and Ashish Motivala. SWIM: Scalable Weakly Consistent Infection-Style Process Group Membership Protocol. In *Proceedings of International Conference on Dependable Systems and Networks*, pages 303–312, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[32] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. Towards

architecture-based self-healing systems. In *WOSS '02: Proceedings of the First Workshop on Self-Healing Systems*, pages 21–26, New York, NY, USA, 2002. ACM Press.

[33] Xavier Défago, Naohiro Hayashibara, and Takuya Katayama. On the Design of a Failure Detection Service for Large-Scale Distributed Systems. In *Procceedings of International 1st Symposium Towards Peta-Bit Ultra-Networks (PBit 2003)*, pages 88 – 95, 2003.

[34] Danny Dolev, Roy Friedman, Idit Keidar, and Dahlia Malkhi. Failure Detectors in Omission Failure Environments. Technical Report 96 - 1608, Department of Computer Science, Cornell University, 1996.

[35] Assia Doudou, Benoit Garbinato, Rachid Guerraoui, and Andre Schiper. Muteness Failure Detectors: Specification and Implementation. In *Dependable Computing - EDCC-3: Proceedings of 3rd European Dependable Computing Conference, Prague, Czech Republic. Lecture Notes in Computer Science*, volume 1667, pages 71 – 87. Springer Berlin / Heidelberg, Sep. 1999.

[36] Lorenzo Falai and Andrea Bondavalli. Experimental Evaluation of the QoS of Failure Detectors on Wide Area Network. In *Proceedings of International Conference on Dependable Systems and Networks (DSN2005)*, pages 624 – 633, Jul. 2005.

[37] Christol Fetzer, Michel Raynal, and Frederic Tronel. An Adaptive Failure Detection Protocol. In *Proceedings of the 8th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'01)*, pages 146 – 153, Seoul (Korea), 2001. IEEE Computer Society Press.

[38] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374 – 382, April 1985.

[39] Ian Foster and Catalin Dumitrescu. VO-Centric Ganglia Simulator. *GriPhyN/iVDGL TechReport*, pages 04–31, 2004.

[40] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1998.

[41] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. In *International Journal of High Performance Computing Applications*, volume 15(3), pages 200 – 222, 2001.

[42] Kumar K. Goswami, Ravishankar K. Iyer, and Luke Young. DEPEND: A Simulation-Based Environment for System Level Dependability Analysis. *IEEE Transactions on Computers*, 46(1):60 – 74, 1997.

[43] Jim Gray. Why Do Computers Stop and What Can We Do About It. In *Proceedings of the 6th International Conference on Reliability and Distributed Databases Systems*, pages 3 – 12, 1986.

[44] Jim Gray. A Census of Tandem System Availability Between 1985 and 1990. *IEEE Transactions on Reliability*, 39(4):409 – 418, 1990.

[45] Object Management Group. The Common Object Request Broker: Architecture and Specification, 1999.

[46] Indranil Gupta, Tushar D. Chandra, and Germán S. Goldszmidt. On Scalable and Efficient Distributed Failure Detectors. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, pages 170 – 179, New York, NY, USA, 2001. ACM Press.

[47] He Hao. What is Service-Oriented Architecture? `http://webservices.xml.com/pub/a/ws/2003/09/30/soa.html`, September 2003.

[48] N. Hayashibara, X.Défago, and T. Katayama. Two-ways Adaptive Failure Detection with The φ Failure Detector. In *Procceedings International Workshop on Adaptive Distributed Systems. In conjunction with 17th International. Symposium on Distributed Computing (DISC-17)*, pages 22 – 27, Oct. 2003.

[49] Naohiro Hayashibara. *Accrual Failure Detectors*. PhD thesis, Japan Advanced Institute of Science and Technology, 2004.

[50] Naohiro Hayashibara, Adel Cherif, and Takuya Katayama. Failure Detectors for

Large-Scale Distributed Systems. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS' 02)*, pages 404 – 409, Washington, DC, USA, 2002. IEEE Computer Society.

[51] Naohiro Hayashibara, Xavier Defago, Rami Yared, and Takuya Katayama. The Accrual Failure Detector. In *Proceddings of 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)*, pages 66 – 78, Florianopolis, Brazil, 2004.

[52] F. Howell and R. McNab. SimJava: A Discrete Event Simulation Package For Java With Applications In Computer Systems Modelling. In *Proceedings of the First International Conference on Web-based Modelling and Simulation*. San Diego, CA, Society for Computer Simulation, 1998.

[53] H.Sanneck. *Packet Loss Recovery and Control for Voice Transmission ver the Internet*. PhD thesis, Technischen Universität Berlin, Berlin, 2000.

[54] Michel Hurfin, Achour Mostefaoui, and Miche Raynal. Consensus in Asynchronous Systems Where Processes Can Crash and Recover. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 280 – 286. IEEE Computer Society, 20-23 Oct 1998.

[55] Foster I., H. Kishimoto, A. Savva, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, et al. The Open Grid Services Architecture, Version 1.0. *Global Grid Forum, Lemont, Illinois, USA, GFD-I*, 30, 2005.

[56] Van Jacobson. Congestion Avoidance and Control. In *ACM SIGCOMM Computer Communication Review*, volume 25(1), pages 157 – 187. ACM Press New York, NY, USA, 1995.

[57] Peter Watts Jones and Peter Smith. *Stochastic Processes: Methods and Applications*. Oxford University Press, US, Sep. 2001.

[58] P. K. Kapur, S. Kumar, and R. B. Garg. *Contribution to Hardware and Software Reliability Modeling*. World Scientific Publishing Company, 1999.

[59] Kim Potter Kihlstrom, Louise E. Moser, and P. M. Melliar-Smith. Solving Con-

sensus in a Byzantine Environment Using an Unreliable Fault Detector. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*, pages 61 – 75, 1997.

[60] Nick Kolettis and N. Dudley Fulton. Software Rejuvenation: Analysis, Module and Applications. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, pages 381 – 390, Washington, DC, USA, 1995. IEEE Computer Society.

[61] Richard Koo and Sam Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Trans. Softw. Eng.*, 13(1):23–31, 1987.

[62] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[63] J.C. Laprie, A. Avizienis, and H. Kopetz. *Dependability: Basic Concepts and Terminology*. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 1992.

[64] Heiko Ludwig, Alexander Keller, Asit Dan, Richard P. King, and Richard Franck. Web Service Level Agreement WSLA Language Specification. www.research.ibm.com/wsla/, Jan. 2002.

[65] D. Manivannan and M. Singhal. A Low-Overhead Recovery Technique Using Quasi Synchronous Checkpointing. In *Proceedings IEEE Int. Conf. Distributed Comput. Syst*, pages 100 – 107, Los Alamitos, CA, USA, 1996. IEEE Computer Society.

[66] Steven McCanne and Sally Floyd. ns-2 Network Simulator. *Obtain via: http://www. isi. edu/nsnam/ns*, 1995.

[67] Kaddour Najim, Enso Lkonen, and Ait-Kadi Daoud. *Stochastic Processes: Estimation, Optimisation and Analysis*. Kogan Page Science, Jul. 2004.

[68] Raul Ceretta Nunes and Ingrid Jansch-Porto. QoS of Timeout-Based Self-Tuned Failure Detectors: The Effects of the Communication Delay Predictor and the Safety Margin. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN2004)*, pages 753 – 761, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[69] Rui Oliveira, Rachid Guerraoui, and André Schiper. Consensus in the Crash-Recover Model. *Lausanne-Switzerland: Départment d'Informatique, EPFL, 45p*, pages 97 – 239, 1997.

[70] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why Do Internet Services Fail, and What Can be Done About It? In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, pages 1– 16, Seattle, WA, USA, 2003.

[71] Robbert Van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. Technical report, Cornell University, Ithaca, NY, USA, 1998.

[72] Kaustubh R.Joshi, Matti A.Hiltunen, and William H.Sanders. Automatic Recovery Using Bounded Partially Observable Markov Decision Processes. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*, pages 445 – 456. IEEE Computer Society Washington, DC, USA, 2006.

[73] Thomas Sandholm and Jarek Gawor. Globus Toolkit 3 Core—A Grid Service Container Framework. `www-unix.globus.org/toolkit/3.0/ogsa/docs/gt3_core.pdf`, 2003.

[74] Ma Schoenfelder and Wa Rogers. A Rainbow Net Simulator with a Dependability Application. In *Proceedings of the Tenth Annual International Phoenix Conference on Computers and Communications*, pages 71 – 77, AZ, USA, 1991.

[75] Bianca Schroeder and Garth A. Gibson. A Large-Scale Study of Failures in High-Performance-Computing Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN2006)*, pages 249 – 258, Philadelphia, PA, USA, 2006. IEEE Computer Society.

[76] R. Sessions. *COM and DCOM: Microsoft's vision for distributed objects*. John Wiley & Sons, Inc. New York, NY, USA, 1997.

[77] Michael E. Shin and Daniel Cooke. Connector-based Self-Healing Mechanism for Components of a Reliable System. In *DEAS '05: Proceedings of the 2005*

workshop on Design and evolution of autonomic application software, pages 1–7, New York, NY, USA, 2005. ACM Press.

[78] Jim Shore. Fail Fast. *IEEE Software*, 21(5):21 – 25, 2004.

[79] H.J. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, K. Taura, and A. Chien. The MicroGrid: A Scientific Tool for Modeling Computational Grids. *Scientific Programming*, 8(3):127 – 141, 2000.

[80] Irineu Sotoma and Edmundo Roberto Mauro Madeira. Adaption - Algorithms to Adaptive Fault Monitoring and Their Implementation on CORBA. In *DOA '01: Proceedings of the Third International Symposium on Distributed Objects and Applications*, pages 219 – 228, Washington, DC, USA, 2001. IEEE Computer Society.

[81] Irineu Sotoma and Edmundo Roberto Mauro Madeira. A Markov Model for Quality of Service of Failure Detectors in the Pressure of Loss Bursts. In *AINA '04: Proceedings of the 18th International Conference on Advanced Information Networking and Applications*, volume 2, pages 62 – 67, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[82] Paul Stelling, Ian Foster, Carl Kesselman, Craig A. Lee, and Gregor von Laszewski. A Fault Detection Service for Wide Area Distributed Computations. *Cluster Computing*, 2(2):117 – 128, 1999.

[83] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maguire, T. Sandholm, P. Vanderbilt, and D. Snelling. Open Grid Services Infrastructure (OGSI) Version 1.0. *Technical Report, Open Grid Services Infrastructure. WG, Global Grid Forum*, 6, 2003.

[84] Peter Urban, Andre Schiper, and Xavier Defago. Neko: A Single Environment to Simulate and Prototype Distributed Algorithms. In *Proceedings of the 15th International Conference on Information Networking (ICOIN-15)*, pages 503–511, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[85] Kalyanaraman Vaidyanathan, Richard E. Harper, Steven W. Hunter, and Kishor S. Trivedi. Analysis and Implementation of Software Rejuvenation in Cluster Sys-

tems. In *SIGMETRICS '01: Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 62–71, New York, NY, USA, 2001. ACM Press.

[86] W3C. XML Schema, W3C Recommendation. `http://www.w3.org/TR/xmlschema-0`, May 2001.

[87] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pe-Yu Chung, and C. Kintala. Checkpointing and its Applications. In *Proceedings of the Twenty Fifth International Symposium on Fault-Tolerant Computing (FTCS-25)*, pages 22 – 31, Los Alamitos, CA, USA, 1995. IEEE Computer Society.

Appendix A

Failure Detection Algorithms

Algorithm 3 NFD-S Algorithm

Process p

1: for some constant η, send to q heartbeat messages $m_1, m_2, m_3, \cdots$ at regular time points $\eta, 2\eta, 3\eta, \cdots$ respectively;

Process q

2: Initialization:
3: **for all** $i \geq 1$, set $\tau_i = \sigma_i + \delta$; **do** $\{\sigma_i = i\eta$ is the sending time of $m_i\}$
4: output $= S$; { suspect p initially }
5: **end for**
6: at every freshness point τ_i:
7: **if** no message m_j with $j \geq i$ has been received **then**
8: output $\leftarrow S$; { suspect p if no fresh messages received is received }
9: **end if**
10: upon receive message m_j at time $t \in [\tau_i, \tau_{i+1})$:
11: **if** $j \geq i$ **then** {trust p when some fresh message is received}
12: output $\leftarrow T$;
13: **end if**

Algorithm 4 NFD-U Algorithm

Process p: { using p's local clock time}

1: for some constant η, send to q heartbeat messages $m_1, m_2, m_3, \cdots$ at regular time points $\eta, 2\eta, 3\eta, \cdots$ respectively;

Process q: { using q's local clock time}

2: Initialization:
3: **for all** $i \geq 1$, set $\tau_i = EA_i + \alpha$; **do** $\{EA_i$ is the expected arrival time of $m_i\}$
4: output $= S$; { suspect p initially }
5: **end for**
6: at every freshness point τ:
7: **if** no message m_j with $j \geq i$ has been received **then**
8: output $\leftarrow S$; { suspect p if no fresh messages received is received }
9: **end if**
10: upon receive message m_j at time $t \in [\tau_i, \tau_{i+1})$:
11: **if** $j \geq i$ **then** {trust p when some fresh message is received}
12: output $\leftarrow T$;
13: **end if**

Algorithm 5 NFD-E Algorithm

Process p: { using p's local clock time}

for some constant η, send to q heartbeat messages $m_1, m_2, m_3, \cdots$ at regular time points $\eta, 2\eta, 3\eta, \cdots$ respectively;

Process q: { using q's local clock time}

Initialization:
$\tau_0 = 0$;
$\ell = -1$; { ℓ keeps the largest sequence number in all messages q received so far }
upon $\tau_{\ell+1}$ = the current time: { if the current time reaches $\tau_{\ell+1}$, then all messages received are stale}
output $\leftarrow S$; { suspect p since all messages received are stale at this time }
upon receive message m_j at time t:
if $j > \ell$ **then** {received a message with a higher sequence number}
    $\ell \leftarrow j$;
    compute $\widehat{EA}_{\ell+1}$; {estimate the expected arrival time of $m_{\ell+1}$ using $\widehat{EA}_{\ell+1} \approx \frac{1}{n}(\sum_{i=1}^{n} A'_i - i \cdot \eta) + (\ell+1)\eta$}
    $\tau_{\ell+1} \leftarrow \widehat{EA}_{\ell+1} + \alpha$;
    if $t < \tau_{\ell+1}$ **then**
        output $\leftarrow T$; { trust p since m_ℓ is still fresh at time t }
    end if
end if

Algorithm 6 Lazy Failure Detection Protocol

1: **when SEND** M to p_j **is invoked**:
2: m.content $\leftarrow$ M; m.st $\longleftarrow$ hc_i;
3: $pending\_msg\_st_i[j] \leftarrow pending\_msg\_st_i[j] \cup \{m.st\}$
4: send appl(m) to p_i
5: **when** type(m) **is received** from p_j:
6: **case** $type = appl$ **then** transmit $M = m.content$ to the upper layer; % **RECEIVE** M %
7: send ack(m) to p_j % m.st keeps its value %
8: type= ack **then** $rt \leftarrow hc_i$;
9: $max\_rtd_i[j] \leftarrow max(max\_rtd_i[j], rt - m.st)$;
10: $pending\_msg\_st_i[j] \leftarrow pending\_msg\_st_i[j] - \{m.st\}$
11: type =ping **then** send ack(m) to p_j % m.st keeps its value %
12: **endcase**
13: **when** QUERY(j) **is invoked**;
14: **if** $pending\_msg\_st_i[j] = \emptyset$ **then**
15: creat a control message m;
16: m.content $\leftarrow$ null; m.st $\leftarrow$ hc_i;
17: send ping(m) to p_j;
18: $pending\_msg\_st_i[j] \leftarrow \{m.st\}$;
19: **return** $(no\_suspect)$
20: **else** $\{rt \leftarrow hc_i;\}$
21: **if** $rt -$ min$(pending_msg_st_i[j]) > max_rtd_i[j]$ **then**
22: **return** (suspect)
23: **else**
24: **return** (no_ suspect)
25: **end if**
26: **end if**

Algorithm 7 φ-Failure Detector

Initialization:

1: $s_p := -1$ {keep the largest sequence number}
2: $WS :=$ α constant {the window size is a constant}
3: $LA_p := 0$ {the arrival time in the previous receipt}
4: $\Delta_{H_p^i} := 0$ {inter-arrival time}
5: $\text{sum}_p := sqr_p := 0$ {summation and std. deviation of $\Delta_{H_p^i}$}

Task 1: {Sampling data}

6: **upon** receive heartbeat H_p^i
7: **if** $i > s_p$ **then**
8: $\quad \Delta_{H_p^i} := A_p^i - LA_p$
9: $\quad LA_p := A_p^i$
10: $\quad s_p := i$
11: $\quad sum_p := \sum_{j=i-(WS-1)}^{i} \Delta_{H_p^i}$
12: $\quad sqr_p := \sum_{j=i-(WS-1)}^{i} (\Delta_{H_p^i})^2$
13: $\quad \sigma_p^2 := \sqrt{\frac{sqr_p}{WS} - (\frac{sum_p}{WS})^2}$
14: $\quad \mu_p := \frac{sum_p}{WS}$
15: **end if**

Task 2: {Calculation for φ}

16: **upon** receive request from p about q at time t
17: $\quad \varphi_p := \varphi(t)$
18: **return** φ_p

Algorithm 8 Bertier *et al.*'s Algorithm

Every process $p \in \Pi$ **executes:**

Initialization:

$\text{suspect}_p \leftarrow \emptyset$

for all $q \in \Pi - \{p\}$ **do**

$\Delta_P(q)= 0$ $\{\Delta_p(q)$ belongs to the second layer and allows to moderate the detection$\}$

$\tau_0(q) = 0$ {Initially, all process q will be suspected by process p}

$EA_{(0)}(q) = U_0(q) = 0$, $\text{delay}_0(q) =$ initial value

$\alpha_0(q) = \text{var}_0(q) = \text{error}_0(q) = 0, A_{(0)}(q) = 0$

$k(q) = -1$ {$k(q)$ keeps the largest sequence number in all the messages p received from q so far}

end for

Task 1:

at time i, η, sends heartbeat m_i to $\Pi - \{p\}$ {η is the detection interval}

Task 2:

upon receive message m_j at time t from q:

if $j > k(q)$ **then** {received a message with a higher sequence number}

$k(q) \leftarrow j$

$\text{error}_k(q) \leftarrow t - EA_k(q) - \alpha_k(q)$

$\text{delay}_{k+1}(q) \leftarrow \text{delay}_k(q) + \gamma.\text{error}_k(q)$

$\text{var}_{k+1}(q) \leftarrow \text{var}_k(q) + \gamma.|(\text{error}_k| - \text{var}_k(q))$

$\alpha_{k+1}(q) \leftarrow \beta.\text{delay}_{(k+1)}(q) + \phi.\text{var}_{(k+1)}(q)$

if $j < n$ **then**

$U_{(k+1)} = \frac{t}{k+1} * \frac{k}{k+1} U_k$, $EA_{k+1} = U_{k+1} + \frac{k+1}{2}.\eta$

else

$EA_{(k+1)} = EA_{(k)} + \frac{1}{k}.(t - A_{(k-n-1)}(q))$

end if

$A_{(k)}(q) \leftarrow t$ {$A(q)$ is an array which contains the n last message arrival dates from q}

$\tau_{(k+1)}(q) \leftarrow \tau_k(q) + EA_{(k+1)}(q) + \alpha_{(k+1)}(q)$ {set the next freshness point $\tau_{\ell+1}(q)$}

if $q \in \text{suspect}_p$ **then**

$\text{suspect}_p \leftarrow \text{suspect}_p - \{q\}$ {trust q since $m_k(q)$ is still fresh at time i}

$\Delta_p(q) \leftarrow \Delta_p(q) + 1$ {increase the timeout period}

end if

end if

Task 3:

Upon $\tau_{k+1}(q) =$ the current time: {if the current time reaches τ_{k+1}, then none of the messages received is still fresh}

wait during $\Delta_p(q)$ and if no message receive from q {detection moderation}

$\text{suspect}_p \leftarrow \text{suspect}_p \cup \{q\}$ {suspect q since no received message is still fresh at this time}